MATHEMATICAL CENTRE TRACTS 86

S.G. VAN DER MEULEN
M. VELDHORST

TORRIX

A PROGRAMMING SYSTEM FOR OPERATIONS ON VECTORS AND MATRICES OVER ARBITRARY FIELDS AND OF VARIABLE SIZE

VOLUME I

MATHEMATISCH CENTRUM AMSTERDAM 1978

AMS(MOS) subject classification scheme (1970): 15-04, 15A03, 15A33,
65F99, 68-02, 68A10.

ISBN 90 6196 152 1

CONTENTS

PREFACE

ABSTRACT AND INTRODUCTION

1. MATHEMATICAL FOUNDATIONS 1

2. LANGUAGE AND IMPLEMENTATION 23

3. USERS GUIDE 55

4. ORGANIZATIONAL MATTERS 129

5. TORRIX BASIS 143

6. BASIS: ROUTINETEXTS 175

INDEX 215

BIBLIOGRAPHY 229

PREFACE

Utrecht, May 1978

This text reports on work done at the Department of Computer Science of the University of Utrecht. We are grateful to the Mathematical Centre in Amsterdam for giving us access to their well-established publication channel for the final version of this book and for financial support.

Two preliminary versions for limited circulation, issued by the University of Utrecht, preceded the present publication: the first dated May 1976, the second February 1977. They now cease to have other than historical value, though the second report contains no technical material that became out of date (apart from the many improvements in the formulation of the routine-texts in chapter 6). This is the place and the time to acknowledge all critical reactions and valuable suggestions of various people who evidently have read these reports and wanted to use TORRIX as a standard-basis for algorithms in numerical analysis and other mathematical applications.

We are greatly indebted to Prof Dr T.J. Dekker of the University of Amsterdam, to Prof Dr P.J. van Houwen of the Mathematical Centre and to our friends on their staffs for their stimulating criticism and wholehearted support which, in no small measure, contributed to the final shape of TORRIX as a programming tool.

We thank Jos Schlichting of CDC Holland for his advice, our guest Joe Kohler for his aid in finding our way through certain difficulties in the natural language part of this book, Tanja Schwarz and Emmy Busch for their skilful and patient typing and retyping the successive manuscripts.

We owe a special kind of gratitude to our former chairman Prof Dr A. van der Sluis for allowing at a critical moment the so essential practical side of this work.

University of Utrecht
Department of Computer Science
Budapestlaan 6
3508 TA Utrecht/Uithof
The Netherlands

S.G. van der Meulen
M. Veldhorst

ABSTRACT AND INTRODUCTION

This is the defining document of a programming system, named TORRIX, for operations on the objects (vectors, matrices etc.) in rather general linear spaces. It is also a detailed report on TORRIX68: the implementation of TORRIX, as far as was possible, in ALGOL68 - being the only well-defined programming language both available and suitable for the purpose. We give an account of this choice in chapter 2.

Our main objective was to find and to design the computational counterpart of the modern algebraic approach to linear vector spaces. Where subroutine libraries for operations on vectors and matrices, in particular for numerical applications, more or less adequately followed the progress in electronic computation from its very beginning, it is remarkable that almost nothing has been done in following - not even inadequately - the development of modern algebra in this area. It is a somewhat strange fact that the approach to vectors and matrices in the environment of computers remained almost entirely on the level of arrays with fixed bounds (usually even to be known at compile time) over the real and/or complex field only.

We were interested in a computational concept in which the scalar system underlying a linear space can be in principle any field or ring or other relevant algebraic system - commutative or skew, infinite or finite, ordered or unordered. Moreover, as a consequence, we would then require a particular program sourcetext - whenever it has a meaning for different such scalar systems - to be invariant over them. In other words: our intended computational counterpart to a mathematical text on abstract vector spaces is a program that can be compiled for different choices of the scalar subsystem. Further, where vectors and their linear transformations (matrices) can be defined as coordinate-free objects, we wanted to have the option of treating them accordingly. Finally, observing that the dimension of a linear space and of all its subspaces is a rather subsidiary parameter, we wanted to deal with it as such. These latter two requirements imply in fact the removal of all bound restrictions in operations on vectors and matrices.

TORRIX is the outcome of this endeavour. The objective of invariance of the program over the scalar subsystem appeared to be mainly a matter of programming language features. Section 2.2 deals with them; in particular 2.2.6 and 2.2.7 may be of interest for future developments. The principle of independence from coordinates and dimension was mainly a matter of data-representation and could be realized by the lucky strike of extending each "concrete" array to a "total" one by completing it with "virtual zeroes". Section 1.2 treats the mathematical justification; section 2.3 deals with the practical realization of this method.

Although TORRIX68, being a particular implementation of TORRIX, rests entirely on ALGOL68, a reader with even less than a nodding knowledge of the parent language can nevertheless be sure that he will have no difficulty in grasping the essence of this book (which is certainly not a text on ALGOL68). TORRIX68 is, to a large extent, a transformation of ALGOL68 into a special purpose language, into which chapter 3 is a complete and rather elementary introduction. Other implementations, for instance as an autonomous language, seem to be quite feasible and somebody might feel like attempting it after reading this report.

The present volume is on TORRIX-BASIS (i.e. the basic operations only). A second volume will follow in due time and treat the application of TORRIX-BASIS to complex (Hermitian) and sparse matrix systems as also to a few other, more specific areas.

For this volume we had roughly three categories of readers in mind: those who just want to know what TORRIX is about (perhaps without even being a programmer), those who want to use TORRIX68, and those who are interested in its implementation.

For the first category we wrote chapter 1.
For the potential users chapter 3 may serve as a guide.
Chapter 2 is a report on the implementation and may also be of interest for computer scientists in the fields of programming language design and of software engineering.
Chapter 5 is a concise reference manual for all three categories.
The more technical chapters 4 and 6 establish the de facto release of the programming system TORRIX68.

Chapters 1, 2 and 3 are quite independent treatises on different aspects of the same subject; they have also been written in different periods of time. Their reading order is immaterial. However, readers who wish to get a quick insight into what it is all about, are advised to start with the introductions to the chapters and their main sections (under headings with one or two digits) in the order in which they are presented, and then to decide where, eventually, to proceed.

1. MATHEMATICAL FOUNDATIONS

1.1	ALGEBRAIC SYSTEMS	4
1.1.1	Monoids, groups, rings and fields	4
1.1.2	Vector spaces	6
1.1.3	Finite sequences, bases and dimension	8
1.1.4	Linear transformations and matrices	9
1.1.5	Ordered systems and innerproduct spaces	10
1.2	TORRIX ARRAYS	11
1.2.1	Representation of scalars	11
1.2.2	Arrays and their equivalence classes	12
1.2.3	Total and concrete arrays	14
1.2.4	Concrete representations of vectors and matrices	15
1.3	TORRIX SYSTEMS	17
1.3.1	Concrete domains	18
1.3.2	Concrete operations	20

1. MATHEMATICAL FOUNDATIONS

Finite sequences play an important role in the vast majority of computer programs. Depending on the language in use, they will be known as "arrays", "rows", "dimensioned values", "subscripted values", "multiple values" etc. We shall adopt the technical term array to denote objects that are or comprise finite sequences. Depending on the application area, arrays may represent vectors, matrices, polynomials, power series, series of measurements or other data, tables or - quite generally - all different kinds of enumerated sets of values on which certain operations have been defined. These operations will be the subject matter of this chapter.

Mathematically, we define an array to be a function with a connected domain in the integers and a codomain (range) of in principle any kind. Due to limitations imposed on most programming systems, domains are then technically restricted to intervals [1:n] - i.e. to intervals in the natural numbers with lowerbound 1 and upperbound n (often to be known "at compile time"). The codomains are usually confined to the (integral or real) numbers and maybe a few more possibilities (complex numbers and/or logical values).

We now want to regard an array as an entity on its own, as one functional object rather than as a set of numbered objects. Moreover, we do not want any a priori restriction on the domain, neither do we want to be needlessly tied down to a specific (numerical) codomain. Our first objective is the construction of a tool: a useable piece of programming equipment for sane (and safe) operations on such rather general entities.

To that purpose we need a firm mathematical foundation. We want to avoid arbitrary operations which may (perhaps) be nice for certain goals, but lack generality and quite often appear to be traps. An appropriate mathematical guarantee for the consistency of our approach will be found in the pure algebraic, coordinate-free concepts of a vector space over an arbitrary field, of a module over an arbitrary ring and of even weaker systems if we need them. These systems cover a wide spectrum of applications - from numerical analysis (linear systems, polynomials, function approximations etc.) and more abstract algebraic manipulations, via statistical computations, various computations in operations research, decision making and system theory, until and including the area of system simulation.

However, taking pure mathematics as our guide, we must be well aware of at least three essential differences between a mathematical and a computational system:

1) Mathematical functions are static (timeless) relations between sets. Computational operations are dynamic - they always carry along two attributes: before and after. They generate, change and destroy information.
2) Mathematical objects have an inherent uniqueness whereas there may be several instances of the same mathematical value in a computer memory (in different locations and possibly also in different representations).
3) Mathematics seeks to represent its objects in a way that demonstrates best the cogency of its arguments and the elegance of its proofs. In a computational environment the decisive criteria are less straightforward. The often conflicting economies of memory size, of storage allocation and of time and money interfere in an often rather nasty manner with the more elevated economy and elegance of mathematical reasoning.

We will therefore find in TORRIX object representations of which mathematicians would never have dreamt. We will also find many operations that have no true counterpart in mathematics, endowed as they are with "before" and "after", and also because they treat the possible polypresence of values in a memory and/or cater for specific demands of computational economy.

1.1 ALGEBRAIC SYSTEMS

In this section we briefly summarize the algebraic systems which underlie TORRIX. For their mathematical contents, properties, use and significance we refer to the litterature (e.g. {17} and {16}). We need them in their abstract dressing for the justification of the typical TORRIX representations and operations, and also to establish terminology and notation.

An algebraic system is a set A together with one or more n-ary operations: $A^n \longrightarrow A$ which have to satisfy specified axioms. For TORRIX we need only to consider nullary operations: $A^0 \longrightarrow A$ (the selection of a specified element, e.g. zero), unary operations: $A \longrightarrow A$ and binary operations: $A \times A \longrightarrow A$. A nullary operation will always be denoted by the element selected. Instead of "unary" and "binary" we shall write monadic and dyadic, because the term "binary" may lead to confusion in a computational environment. Fundamental and well-known algebraic systems are:

$\mathbb{N}$	the natural numbers	$\{0,1,2,\ldots\}$
$\mathbb{Z}$	the integral numbers	$\{0,\pm1,\pm2,\ldots\}$
$\mathbb{Z}_n$	the integral numbers modulo n	$\{0,1,\ldots,n-1\}$
$\mathbb{Q}$	the rational numbers	$\{\pm\frac{m}{n} \mid m,n\in\mathbb{N}, n\neq 0\}$
$\mathbb{R}$	the real numbers, i.e. the analytic completion of $\mathbb{Q}$	
$\mathbb{C}$	the complex numbers, i.e. the complexification of $\mathbb{R}$	

1.1.1 Monoids, groups, rings and fields

A semigroup $(S,\square)$ is a set S together with a dyadic operation $\square$: $S \times S \longrightarrow S$ which is associative. A monoid $(M,\square,n)$ is a semigroup with a neutral element $n\in M$, i.e. an element with the property that for all $a\in M$ we have $a\square n = n\square a = a$. We call a monoid $(M,+,0)$ additive and a monoid $(M,\times,1)$ multiplicative. An additive monoid is commutative (unless specified otherwise), a multiplicative monoid may or may not be commutative.

In mathematical texts the operator symbol "$\times$" is usually omitted (but never in a programming language). We thus have:

$a+(b+c)=(a+b)+c$	semigroup	$a(bc)=(ab)c$
$0+a=a+0=a$	monoid	$1a=a1=a$
$a+b=b+a$		

For $a_1+\ldots+a_n$ in an additive monoid we often write $\sum_n a_i$.
For $a_1\times\ldots\times a_n$ in a multiplicative monoid we often write $\prod_n a_i$.
If all $a_i=a$, we write na for $\sum^n a$ and a^n for $\prod^n a$ Identities like $(m+n)a = ma+na$, $a^m a^n=a^{m+n}$, $m(na)=(mn)a$ and $(a^m)^n=a^{mn}$ are quite obvious.

A group $(G,\square,n,')$ is a monoid together with a monadic operator inverse, denoted by $'$. Hence, a group is a monoid in which there exists an inverse $a'\in G$ for every $a\in G$, such that $a\square a'=a'\square a=n$. In a (commutative or abelian) additive group we write $-a$ for the inverse and $a-b$ for $a+-b$. In a multiplicative group we write a^{-1} for the inverse and a/b for $a\times b^{-1}$, hence $a^{-1}=1/a$.

We thus have:

in an (abelian) additive group: $a-a=0$
in a multiplicative group: $aa^{-1}=a^{-1}a=1=a/a$

(it can be proved that in a non commutative group a left inverse and a right inverse are equal and unique).

A ring is a combination of an abelian group $(R,+,0,-)$ and a multiplicative monoid $(R,\times,1)$ into one system $(R,+,0,-,\times,1)$ in which multiplication is distributive over addition:

$a(b+c)=ab+ac$ and $(a+b)c=ac+bc$

A commutative ring is a ring in which the multiplication is also commutative. If $a,b\in R$, $a\neq 0$ and $b\neq 0$, but nevertheless $ab=0$, we call a and b zero divisors. In a ring without zero divisors the cancellation law holds:

if $ax=ab$ and $a\neq 0$ then $x=b$
if $xa=ba$ and $a\neq 0$ then $x=b$

A ring has no zero divisors iff the cancellation law holds.

A field F is a ring in which the subset $F\backslash\{0\}$ of non-zero elements is a multiplicative group - i.e. in which every $a\neq 0$ has a multiplicative inverse $1/a$. We call a non-commutative field (i.e. a field with a non-commutative multiplication) a skew field, but normally assume a field to be commutative.

The fundamental systems $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}_n$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ all combine a commutative additive monoid with a commutative multiplicative monoid in such a manner that multiplication is distributive over addition. In $\mathbb{N}$ neither of the monoids is a group, hence $\mathbb{N}$ is not a ring. In $\mathbb{Z}$, $\mathbb{Z}_n$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ the additive monoids are groups, therefore these systems are rings. $\mathbb{Z}_n$ has zero-divisors unless n happens to be prime - i.e. $\mathbb{Z}_p$ (p prime) has no zero-divisors. In $\mathbb{Z}$ the multiplication has no inverse, hence $\mathbb{Z}$ is not a field; the same applies to $\mathbb{Z}_n$ (n not prime), but $\mathbb{Z}_p$ is a (finite) field. $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ are fields.

One may, starting from the Peano axioms, construct $\mathbb{Z}$ from $\mathbb{N}$, $\mathbb{Q}$ from $\mathbb{Z}$, $\mathbb{R}$ from $\mathbb{Q}$ and finally $\mathbb{C}$ from $\mathbb{R}$. In these constructions the mother system is always isomorphic to a subset of its daughter. Apart from these rather formal isomorphisms we thus have:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C} \qquad \text{and also} \qquad \mathbb{Z}_n \subset \mathbb{Z}$$

Another and more straightforward way of looking at these inclusions is that $\mathbb{R}$ can be obtained from $\mathbb{C}$ by leaving out the imaginary (parts of) numbers, $\mathbb{Q}$ from $\mathbb{R}$ by leaving out the irrational numbers, $\mathbb{Z}$ from $\mathbb{Q}$ by leaving out all fractions and $\mathbb{N}$ from $\mathbb{Z}$ by leaving out the negative integers. This will be the way we shall look at such inclusions.

1.1.2
Vector spaces

Vector spaces are built on a field F, the elements of which are usually called scalars. This F may, eventually, be restricted to a ring R, or to an even weaker system by leaving out certain operations and/or elements. It will then be tacitly assumed that the vector space can (and will) be restricted accordingly. In other words: though we shall mostly speak of vector spaces, we may also have weaker systems in our mind. Moreover, we shall assume the underlying field to be commutative in order to avoid tedious distinctions in left- and right operations. TORRIX, however, is not confined to commutative fields.

The abstract notion of a vector space V over a field F comprises a set of elements, called vectors, satisfying the axioms:

1) V is a commutative additive (abelian) group.

2) V admits the scalars of F as linear operators.

We shall write u,v,w,... for vectors and denote the scalars by small Greek letters $\alpha,\beta,\gamma,\ldots\lambda,\kappa,\mu,\ldots\upsilon,\phi,\psi,\omega,\sigma,\eta\ldots$

We thus have:

1)	$u+(v+w)=(u+v)+w$	(associativity)
	$u+v = v+u$	(commutativity)
	$u+o = u$	(existence unique <u>zerovector</u> o)
	$u-u = o$	(existence unique <u>inverse</u> -u)
2)	$\alpha(u+v) = \alpha u+\alpha v$	(distributivity over vectors)
	$(\alpha+\beta)u = \alpha u+\beta u$	(distributivity over scalars)
	$(\alpha\beta)u = \alpha(\beta u)$	(associativity)
	$1u = u$	(scalar unity, identity operator)

More general linear operators on V are the (left and right) <u>linear transformations</u> L: $V\longrightarrow V$ and R: $V\longrightarrow V$ which map every vector $v\in V$ into a vector $Lv\in V$ or $vR\in V$. The defining property of linear transformations is that $L(\alpha u+\beta v) = \alpha Lu+\beta Lv$ and $(u\alpha+v\beta)R = uR\alpha+vR\beta$. Where the use of left- or right linear transformations is merely a matter of notational convention (comparable with left or right traffic in different places in the world), we shall drive on the left. TORRIX allows both.

For linear transformations A and B a <u>sum</u> A+B is defined by $(A+B)v = Av+Bv$ and we have a unique <u>zero</u>-transformation O which transforms every $v\in V$ into $o\in V$ - i.e. Ov=o. It is easy to recognize that for this addition of linear transformations both 1) and 2) hold as they hold for vectors:

3a) The set of linear transformations L on a vector space V is itself a vector space over the same field F.

A <u>product</u> of linear transformations is defined as functional composition by $(AB)v = A(Bv)$ and we have a unique <u>identity</u>-transformation I which transforms every $v\in V$ into itself - i.e. Iv=v. This product is a teaser, having two not so nice properties: it is <u>not commutative</u> (not even when the underlying field is) and it admits <u>zero-divisors</u>. The following rules, however, hold - i.e. L forms a non-commutative ring with zero-divisors:

3b) $(AB)v=A(Bv)$, $(AB)C=A(BC)$ (associativity)

$A(B+C)=AB+AC$, $(A+B)C=AC+AB$ (distributivity)

$AO = OA = O$, $AI = IA = A$ (zero and identity)

If the underlying field is confined to a ring R, we may also speak of a module instead of a vector space.

1.1.3

Finite sequences, bases and dimension

A finite sequence $[1:n]\rightarrow A$ or n-tuple of elements a_i in any algebraic system A will be denoted by $(a_1,\ldots,a_n)$ or briefly (a_i). Hence $(a_1,\ldots,a_n)=(a_i)\in A^n$. One should not confuse a finite sequence with a TORRIX array (cf. 1.2.2).

We call a vector $v\in V$ a linear combination of the vector sequence (u_i) iff there exists a scalar sequence (α_i) such that $v=\sum\limits^{n}\alpha_i u_i$. A vector sequence with the property that none of its linear combinations yields o unless all $\alpha_i=0$ - i.e. unless (α_i) is the zero-sequence $(0,\ldots,0)$ - is called a linearly independent sequence. An arbitrary, possibly infinite, vector set $B\subset V$ is linearly independent iff all its finite subsequences have this property.

We call $B\subset V$ a basis in V iff B is linearly independent and every vector $v\in V$ is a linear combination of a sequence in B. We call V finite-dimensional iff it has a finite basis.

The number of elements in any basis in a finite dimensional vector space is the same as in any other basis; this number is called the dimension of V.

The above summary culminates in the following theorem:

> Every finite, n-dimensional vector space V
> over a field F is isomorphic to F^n

This implies that the sequence (e_i) with $e_i=(0,\ldots,\overset{\overset{i}{\downarrow}}{1},\ldots,0)$ forms a basis in V and that every vector $u\in V$ can be written as $u=(\upsilon_1,\ldots,\upsilon_n)=\sum\limits^{n}\upsilon_i e_i$. In this representation, the vector operations 1) and 2) take the form:

1') $u\pm v = (\upsilon_1,\ldots,\upsilon_n)\pm(\phi_1,\ldots,\phi_n) = (\upsilon_1\pm\phi_1,\ldots,\upsilon_n\pm\phi_n)$

2') $\alpha u = \alpha(\upsilon_1,\ldots,\upsilon_n) = (\alpha\upsilon_1,\ldots,\alpha\upsilon_n)$

So we are back at where we started: finite sequences can represent vectors. In the sequel we shall assume all our vector spaces to be finite dimensional.

Observe that the concepts of linear independence, basis and dimension break down for modules over a ring with zero-divisors as also for weaker systems. Therefore, we could take 1') and 2') as the definition of $u \pm v$ and αu, rather than derive them from 1) and 2). However, we shall see that we have a much better representation for computational purposes (see 1.2.2).

1.1.4

Linear transformations and matrices

A double-subscripted sequence or mn-matrix $[1:m] \times [1:n] \longrightarrow F$ can be denoted by a rectangular scheme:

$$\begin{pmatrix} \alpha_{11}, \dots\dots\dots, \alpha_{1n} \\ \vdots \qquad\qquad \vdots \\ \alpha_{m1}, \dots\dots\dots, \alpha_{mn} \end{pmatrix} \quad \begin{array}{l} m \text{ rows} \\ n \text{ columns} \end{array}$$

or briefly by (α_{ij}).

A linear transformation $A: V \longrightarrow V$ may be represented by a nn-matrix (n being the dimension of V) or square matrix $A=(\alpha_{ij})$. The transformed vector $v=Au$ is then given by:

3') $\quad (\phi_h) = (\sum_i^n \alpha_{hi} \upsilon_i)$

For the more general linear transformations $A: F^n \longrightarrow F^m$, the same formula holds with a mn-matrix. For the sum A+B and product AB of matrices representing linear transformations, we obtain the following rules:

3') $\quad (\alpha_{hk}) \pm (\beta_{hk}) = (\alpha_{hk} \pm \beta_{hk})$

$\quad\quad (\alpha_{hi}) \times (\beta_{ik}) = (\sum_i^n \alpha_{hi} \beta_{ik})$

The obvious constraints on these formulae are that A and B in A+B must be both mn-matrices, whereas if A in AB is a mn-matrix then B must be a np-matrix. We shall see that, in a better representation, we can free ourselves of such restrictions (see 1.2.3 and 1.2.4).

1.1.5

Ordered systems and innerproduct spaces

The fundamental systems N, Z, Q and R are (linearly) ordered and the same may be the case for other algebraic systems:

For all $\alpha,\beta,\gamma\in R$ a less-equal relation $\leq$ exists such that:

either $\alpha\leq\beta$ or $\beta\leq\alpha$ or both,

$\alpha\leq\alpha$, $\alpha\leq\beta$ and $\beta\leq\alpha$ imply $\alpha=\beta$, $\alpha\leq\beta$ and $\beta\leq\gamma$ imply $\alpha\leq\gamma$,

$\alpha\leq\beta$ implies $\alpha+\gamma\leq\beta+\gamma$, $\alpha\leq\beta$ and $0\leq\gamma$ imply $\gamma\alpha\leq\gamma\beta$.

Z_p, Z_n, C and many other systems do not admit an ordering in accordance with the above rules.

A vector space over an ordered field F (or ring) may become an innerproduct space by defining - in addition to 1), 2) and 3) - a scalar-valued function $<,>: V\times V\longrightarrow F$ with the following properties:

4) $\qquad <u,v> = <v,u> \qquad$ (commutativity)

$\qquad <\alpha u+\beta v,w> = \alpha<u,w>+\alpha<v,w> \qquad$ (distributivity)

$\qquad <u,u> \geq 0 \; , \; <u,u>=0$ iff $u=o \qquad$ (metrizability)

Specifically when F is the real system R, the innerproduct space is called the Euclidean space. When we extend R to C, we may again define $<,>: V\times V\longrightarrow R$ by dropping the commutativity and proclaim $<u,v>=\overline{<v,u>}$ where $\bar{\alpha}$ is the complex conjugate of α. A thus defined innerproduct space over C is called a unitary space.

In a finite dimensional vector space the most common realization of $<,>$ is by:

4') $\qquad <u,v> = <(\upsilon_i),(\phi_i)> = \sum^n \upsilon_i\bar{\phi}_i$

which explains the name innerproduct.

1.2 TORRIX ARRAYS

By "TORRIX" we denote a computational system for sequences built on some (presupposed) other system in which operations +,-,0,x,1,.. etc. provide for a suitable arithmetic. There are as many TORRIX systems $\mathcal{T}$ as there are underlying systems $\mathcal{S}$.

Basically, $\mathcal{T}$ consists of the scalars from $\mathcal{S}$ together with two classes of scalar-arrays. Operations, based on the $\mathcal{S}$-arithmetic, have been defined on these arrays so that - after certain provisions - $\mathcal{T}$ yields a vector space when $\mathcal{S}$ yields a field. $\mathcal{S}$ may also yield a ring or another useful algebraic system, in which case then $\mathcal{T}$ yields a module or some other vector-space-like system.

This wording has been chosen with some care. The "arrays" themselves are not the vectors or matrices, they rather supply the basic material - the "certain provisions" are essential (see 1.2.2 and 1.2.3). Being computational systems, $\mathcal{S}$ and $\mathcal{T}$ rather "yield" than "are" algebraic systems: their operations are firmly bound to the representations of their operands and these representations, in their turn, are approximations of mathematical ideals. The most important point, however, is the relation between $\mathcal{S}$ and $\mathcal{T}$: the choice of $\mathcal{S}$ indeed determines the properties of $\mathcal{T}$ (cf. 1.3).

In this section we mainly go into matters of representation. In the following section we shall consider the operations in more detail.

1.2.1 Representations of scalars

On most computers we have available two systems $\mathbb{Z}'\subset\mathbb{Z}$ and $\mathbb{R}'\subset\mathbb{R}$. Both $\mathbb{Z}'$ and $\mathbb{R}'$ are finite: $\mathbb{Z}'$ is a connected interval $[m_-:m_+]\subset\mathbb{Z}$ and $\mathbb{R}'$ is a discrete subset in $\mathbb{R}$ (the "floating point" approximation of the real number system). This subset $\mathbb{R}'$ and its properties form an important chapter in numerical analysis - we only mention that there may be more approximations in different precisions. We shall tacitly assume $\mathbb{Z}'\subset\mathbb{R}'$, so that, in particular, the zeroes and ones of $\mathbb{Z}'$ and $\mathbb{R}'$ coincide (i.e. 0=0 and 1=1) - be it, perhaps, in different representations.

The other fundamental systems Z_n, Q and C are normally not hardware available in any (truncated or approximated) representation. They can, however, easily be realized through subroutines: Z_n as the interval $[0:n-1]\subset Z$, Q' as Z'×Z' and C' as R'×R'. The specific operations to be provided for them follow quite straightforward from those in Z' and R' respectively. The approximation C' of the complex number system will be found in many standard subroutine libraries and normally indeed as an extension R'×R' of R'. In our implementation we shall treat C' in this way.

The realization of possible other systems - (skew) fields, polynomial rings etc. - may technically give more problems. However, once they have been realized, they determine a TORRIX system in precisely the same way as R', Z', Z_n, Q' and C' do. With them again, we always assume $Z'\subset S$ - at least in the sense that $0,1\in Z'$ coincide with $0,1\in S$.

In the sequel we shall normally not distinguish Z' from Z, R' from R, C' from C or Q' from Q. Vector spaces over R and C will be denoted by V and W respectively. Observe, however, that the precision of V and W depends on the precision of the representations R' and C'. Finally, where we realize C' through R'×R', we have $R'\subset C'$ and consequently also $V\subset W$.

1.2.2 Arrays and their equivalence classes

We distinguish in TORRIX two classes of arrays - "array1s" and "array2s"#:

$$\text{array1:}\quad [m:n] \longrightarrow S \quad \text{where} \quad m,n\in Z$$
$$\text{array2:}\quad [p:q]\times[m:n] \longrightarrow S \quad \text{where } p,q,m,n\in Z$$

We shall denote array1s by $[\upsilon_i]$ and array2s by $[\alpha_{ij}]$. Their domains will be denoted by $[\![\upsilon_i]\!]$ and $[\![\alpha_{ij}]\!]$, hence $[m:n]=[\![\upsilon_i]\!]\subset Z$ and $[p:q]\times[m:n]=[\![\alpha_{ij}]\!]\subset Z\times Z$.

The use of the terms "1-dimensional" and "2-dimensional" arrays - in vogue in the programming crowd (including the authors of the ALGOL68 report) - is an ill-considerate abuse of language. The number of subscripts in an array has nothing to do with the dimension of the object it represents. In particular in the context of TORRIX, such terminology would be very misleading. The better terms are "single-subscripted" and "double-subscripted" arrays, which we abbreviate to "array1" and "array2".

The extension of the domains from certain intervals in N or N×N to in principle all intervals in Z or Z×Z respectively, is not the true difference between the concepts of finite sequences and arrays. We shall define equivalence classes of arrays to form the objects proper and we shall also adhere a meaning to empty arrays.

> Two arrays are equivalent
> iff: 1) they are equal in the intersection of their domains,
> 2) they are zero anywhere else.

In mathematicians cant:

$$[\upsilon_i] \cong [\phi_i]$$
$$\text{iff: 1) } \upsilon_i=\phi_i \text{ for all } i\in[\![\upsilon_i]\!]\cap[\![\phi_i]\!]$$
$$\text{2) } \upsilon_i=0 \text{ for all } i\in[\![\upsilon_i]\!]\setminus[\![\phi_i]\!]$$
$$\phi_i=0 \text{ for all } i\in[\![\phi_i]\!]\setminus[\![\upsilon_i]\!]$$

$$[\alpha_{ij}] \cong [\beta_{ij}]$$
$$\text{iff: 1) } \alpha_{ij}=\beta_{ij} \text{ for all } (i,j)\in[\![\alpha_{ij}]\!]\cap[\![\beta_{ij}]\!]$$
$$\text{2) } \alpha_{ij}=0 \text{ for all } (i,j)\in[\![\alpha_{ij}]\!]\setminus[\![\beta_{ij}]\!]$$
$$\beta_{ij}=0 \text{ for all } (i,j)\in[\![\beta_{ij}]\!]\setminus[\![\alpha_{ij}]\!]$$

We shall denote the equivalence classes of $[\upsilon_i]$, $[\phi_i]$, .. by $\underline{u}$, $\underline{v}$, .. and those of $[\alpha_{ij}]$, $[\beta_{ij}]$, .. by $\underline{A}$, $\underline{B}$, .. One easily recognizes that and how operations on $\underline{u}$, $\underline{v}$, .., $\underline{A}$, $\underline{B}$, .. can be defined in order to make them satisfy the axioms required for a vector space, or a module or some such (see also 1.2.4).

As a direct consequence of the above definitions, we can now extend the definition of an array to that of an array over an empty domain - i.e. an empty array:

> The empty array1 belongs to the class $\underline{o}$ of all array1s with zero-elements only. The empty array2 belongs to the class $\underline{O}$ of all array2s with zero-elements only.

The concept of empty arrays appears to be of great practical value.

1.2.3
Total and concrete arrays

The idea of taking the equivalence classes $\underline{u}$,..,$\underline{A}$,.. to represent vectors and matrices, rather than the arrays themselves, emerged from the following consideration:

In the isomorphism $V \Leftrightarrow F^n$ it is merely a matter of convention (convenience) to write a vector $u \in V$ as $(\upsilon_1,\ldots,\upsilon_n) \in F^n$. For instance, $[\upsilon_0,\ldots,\upsilon_{n-1}]$ or $[\upsilon_{k+1},\ldots,\upsilon_{k+n}]$ would have done equally well - even with $k<-1$. In other words: instead of denoting the basis of V by $(e_1,\ldots,e_n)$, we might also choose $[e_{k+1},\ldots,e_{k+n}]$ for any $k \in Z$. Now let T be an immense-dimensional vector space spanned by $[e_{-t},\ldots,e_0,\ldots,e_t]$ with $t \in N$ and t very large - the dimension of T is thus 2t+1. Let V be a proper subspace $V \subset F$. Any vector $u \in V$ can now be conceived as a vector in T:

$$u = [\underset{\underset{-t}{\downarrow}}{0},\ldots\ldots,\underset{\underset{k}{\downarrow}}{0},\upsilon_{k+1},\ldots,\upsilon_{k+n},\underset{\underset{k+n+1}{\downarrow}}{0},\ldots\ldots,\underset{\underset{t}{\downarrow}}{0}]$$

We shall call such arrays in T <u>total arrays</u>:

$$\left.\begin{array}{ll} \underline{\text{total array1}}: & [-t:t] \rightarrow S \\ \underline{\text{total array2}}: & [-t:t]\times[-t:t] \rightarrow S \end{array}\right\} t \in N,\ t \text{ very large}$$

A total array is much too long to be realizable in a computer memory - it would also be a waste of space because most of its elements are zero. However, provided that the dimension of V keeps within bounds, there will be short enough arrays in its equivalence class. We call the realization of such an array a <u>concrete array</u>.

We thus arrived at the following position:

> in TORRIX we manipulate <u>concrete arrays</u> of two kinds: <u>array1s</u>: $[\upsilon_i]$,.. and <u>array2s</u>: $[\alpha_{ij}]$,.. .
>
> the arrays of T can be partitioned in equivalence classes: $\underline{u}$,.. and $\underline{A}$,.. .
>
> in each equivalence class $\underline{u}$ or $\underline{A}$ we define a particular <u>total array</u> representing the class uniquely .

A simpler way of saying this is:

> all <u>concrete arrays</u> will be thought of as being <u>embedded</u> in <u>total arrays</u> .

In practice, of course, we aim at the shortest possible concrete arrays. In particular vectors and matrices belonging to proper subspaces $\mathcal{U}$ of our concrete $\mathcal{V}\subset\mathcal{T}$ can (and always should) be represented by shorter concrete arrays than those needed for $\mathcal{V}$. Observe that the shortest concrete array of $\underline{o}$ and $\underline{O}$ respectively, are the empty array1 and the empty array2.

1.2.4

Concrete representations of vectors and matrices

It is an almost trivial exercise to prove that the equivalence classes of arrays in $\mathcal{T}$ establish a vector space (or module or some such) after defining the right operations for them. It would be a mathematical insult to spell such out. Suffice it to give the basic operations satisfying the axioms 1), 2) and 3) in 1.1.2 and 4) in 1.1.5, and to add just a few remarks.

Using the notation of 1.2.2 and writing $[\![\alpha_{ij}]\!]_1$ for the projection $[p{:}q]$, and $[\![\alpha_{ij}]\!]_2$ for the projection $[m{:}n]$ in $[\![\alpha_{ij}]\!] = [p{:}q]\times[m{:}n]$, we define:

1) $[\omega_i] = [\upsilon_i] \pm [\phi_i]$ $\qquad \boxed{\underline{w} = \underline{u}\pm\underline{v}}$

$\omega_i=\upsilon_i\pm\phi_i$	for	$i\in[\![\upsilon_i]\!]\cap[\![\phi_i]\!]$
$\omega_i=\upsilon_i$	for	$i\in[\![\upsilon_i]\!]\setminus[\![\phi_i]\!]$
$\omega_i=\ \pm\phi_i$	for	$i\in[\![\phi_i]\!]\setminus[\![\upsilon_i]\!]$
$\omega_i=0$	for	$i\notin[\![\upsilon_i]\!]\cup[\![\phi_i]\!]$

2) $[\omega_i] = \alpha[\upsilon_i]$ $\qquad \boxed{\underline{w} = \alpha\underline{u}}$

$\omega_i=\alpha\upsilon_i$	for	$i\in[\![\upsilon_i]\!]$
$\omega_i=0$	for	$i\notin[\![\upsilon_i]\!]$

3) $[\phi_h] = [\alpha_{hi}][\upsilon_i]$ $\qquad \boxed{\underline{v} = \underline{A}\underline{u}}$

$\phi_h=\sum\alpha_{hi}\upsilon_i$	for	$i\in[\![\alpha_{hi}]\!]_2\cap[\![\upsilon_i]\!]$
	and	$h\in[\![\alpha_{hi}]\!]_1$
$\phi_h=0$	for	$h\notin[\![\alpha_{hi}]\!]_1$

$[\gamma_{hk}] = [\alpha_{hk}] \pm [\beta_{hk}]$ $\boxed{\underline{C} = \underline{A} \pm \underline{B}}$

$\gamma_{hk} = \alpha_{hk} \pm \beta_{hk}$ for $(h,k) \in \llbracket \alpha_{hk} \rrbracket \cap \llbracket \beta_{hk} \rrbracket$

$\gamma_{hk} = \alpha_{hk}$ for $(h,k) \in \llbracket \alpha_{hk} \rrbracket \setminus \llbracket \beta_{hk} \rrbracket$

$\gamma_{hk} = \quad \pm \beta_{hk}$ for $(h,k) \in \llbracket \beta_{hk} \rrbracket \setminus \llbracket \alpha_{hk} \rrbracket$

$\gamma_{hk} = 0$ for $(h,k) \notin \llbracket \alpha_{hk} \rrbracket \cup \llbracket \beta_{hk} \rrbracket$

$[\gamma_{hk}] = [\alpha_{hi}][\beta_{ik}]$ $\boxed{\underline{C} = \underline{AB}}$

$\gamma_{hk} = \sum_i \alpha_{hi} \beta_{ik}$ for $i \in \llbracket \alpha_{hi} \rrbracket_2 \cap \llbracket \beta_{ik} \rrbracket_1$
and $(h,k) \in \llbracket \alpha_{hi} \rrbracket_1 \times \llbracket \beta_{ik} \rrbracket_2$

$\gamma_{hk} = 0$ for $(h,k) \notin \llbracket \alpha_{hi} \rrbracket_1 \times \llbracket \beta_{ik} \rrbracket_2$

4) $\sigma = \langle [\upsilon_i], [\phi_i] \rangle$ $\boxed{\sigma = \langle \underline{u}, \underline{v} \rangle}$

$\sigma = \sum \upsilon_i \overline{\phi}_i$ for $i \in \llbracket \upsilon_i \rrbracket \cap \llbracket \phi_i \rrbracket$

$\overline{\phi}_i$ being the complex conjugate of ϕ_i

In plain language everything comes down to regarding the operands as objects from their own individual spaces, say $\mathcal{X}$ and $\mathcal{Y}$. The computation is then performed in $\mathcal{X} \cup \mathcal{Y}$ or $\mathcal{X} \cap \mathcal{Y}$, depending on the operation under consideration. The justification of the given arithmetic lies in the fact that both $\mathcal{X}$ and $\mathcal{Y}$ are proper subspaces of the total space $\mathcal{T}$.

Observe how we actually freed ourselves from all constraints on the domains of the arrays involved. All operations are well-defined for all operands, regardless of their domains.

Where possible we shall avoid the distinction between concrete and total arrays, their equivalence classes and the vectors or matrices they represent. Depending on the context we shall denote these "$\mathcal{T}$-objects" by u,v,..., A,B,... or $[\upsilon_i],[\phi_i],\ldots,[\alpha_{ij}],[\beta_{ij}],\ldots$ etc. and speak freely of "vectors", "matrices" or "arrays".

In the practice of programming, however, we must be well aware of the distinction between total- and concrete arrays. The former are mathematical idealizations, the latter materialized objects. This distinction plays an important role where different computations for the same mathematical operation are possible.

1.3 TORRIX SYSTEMS

For the definition of a TORRIX system we must become more precise in what we mean by the realization of a concrete array in a computer memory. Mathematically, a concrete array is a partial function $D \rightarrow S$ where $D=D_1=I$ or $D=D_2=I\times J$ with I and J intervals $\subset Z$. Our definition of equivalence classes of such functions allowed us to extend them to in principle total functions $Z \rightarrow S$ or $Z\times Z \rightarrow S$ by assigning the S-value zero to all i or (i,j) not in D (virtual zeroes). For these total arrays we defined certain basic operations which made them satisfy the axioms of a vector space.#

None of these definitions can actually decide how an array $D \rightarrow S$ should be realized in a computational environment. It may very well be that a functional description is available. A Hilbert matrix $H=[\eta_{ij}]$ for example, might be given by a functional procedure returning $\eta_{ij}=1/(i+j)$. Such a procedure would then represent its matrix in an almost perfect manner. Even the derivation of pure functional procedures for κH, Hu, H±A etc. is feasible, provided that u and A also obey functional descriptions. However, the vast majority of our arrays comes from measurements (i.e. from input) of which at most very global facts may be known in advance.

Therefore, the rules 1,2,3,4) in 1.2.4 not only strongly suggest, but even practically imply that the individual assignments $i \rightarrow \upsilon$ or $(i,j) \rightarrow \alpha$, of S-values to D-values in fact h a v e b e e n m a d e. So we are led to a concrete domain D actually present in memory as a neatly arranged set of <u>locations</u> wherein we find the instances of S-values assigned to them. Observe that one and the same S-value may show up in different locations as also in different domains.

This kind of actual presence of D implies more than the availability of a sufficient number of locations for scalars. It also implies all information concerning the concrete domain bounds and the physical allocation of the scalars. In our functional objects $D \rightarrow S$ the domain D is at least as important as the scalars assigned to it. Several TORRIX operations even apply to D only, ignoring the codomain entirely. Therefore we first discuss D and

We confined ourselves to intervals [-t:t] in order to avoid needless transfinite reasoning. In the practice of computation there is, of course, an upperbound for the subscripts of all concrete domains which may occur in a program. Hence, we do not lose anything by this confinement.

its locality in a memory. Proceeding from there we arrive quite naturally at the concrete operations on arrays, and we shall see how the mathematical operations +, −, ×, / etc. split up in "generating" and "assigning" versions and how various other operations become important.

1.3.1
Concrete domains

A TORRIX memory consists of a finite but (supposedly) always large enough set $\mathbb{L}$ of locations $\mathbb{l}$ for scalars. In different locations we may find (an instance of) the same scalar, but in one and the same location is place for precisely one scalar at a time - i.e. at any moment the state of $\mathbb{L}$ is given by a function Σ: $\mathbb{L} \longrightarrow \mathcal{S}$. Moreover, also at any moment during a computation, the relevant part of $\mathbb{L}$ will be subdivided in regions described by one or more domains: $\mathbb{L}(D)$ = the region described by D (or the region underlying D). Hence, $\mathbb{L} = \{\mathbb{L}(D) \mid D$ is the concrete domain of an array1 or array2 in memory$\}$.

The domain of a particular conceivable concrete array1 u or array2 A will be denoted by D_u or D_A respectively. We now consider:

$$\mathcal{D}_1 = \{D_u \mid u \text{ concrete array1}\}$$
$$\mathcal{D}_2 = \{D_A \mid A \text{ concrete array2}\}$$
$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$$

The structure of $\mathcal{D}$ is far from simple. First of all we have to distinguish concrete domains which have been realized in memory, and concrete domains which can be generally conceived; clearly the former is a proper subset of the latter: $\Delta_1 \subset \mathcal{D}_1$, $\Delta_2 \subset \mathcal{D}_2$ and $\Delta \subset \Delta_1 \cup \Delta_2$. Hence, $\mathbb{L} = \{\mathbb{L}(D) \mid D \in \Delta\}$. Further, two independent systems of ordering relations play a role: one with respect to the underlying regions and one with respect to the subscript bounds.

A particular vector-domain may describe a row, or a column, or a diagonal of a matrix-domain; it may be also a restriction, an extension or a shift of another domain; a vector-region may be described by a matrix-domain (a vector being considered as a one-row or one-column matrix) etc. This may then lead to at first sight very confusing statements such as "different domains may coincide" (meaning that the same region may be described by different domains) and "equal domains may have an empty intersection" (meaning that different regions may be described by the same domain). One should be well aware of the realities in a computational environment: that different vectors may be defined on the same domain (but then necessarily in different regions) and that the same region may underlie different domains (so that the same assignment of scalars to a region may define different vectors).

Rather than giving a full analysis of the possible relations in $\mathcal{D}$ with respect to memory allocation on the one side and to vectors and matrices on the other side, and of all the interrelations - though interesting enough -, we confine ourselves to the precise definition of *inclusion* and *incidence* of domains. These are the two concepts that play an important role behind the screens in TORRIX:

The *inclusion* of domains depends on their bounds and their type ($\mathcal{D}_1$ or $\mathcal{D}_2$). We shall say that:

$D' \leq D''$ iff:

1) they are of the same type (both $\in\mathcal{D}_1$ or both $\in\mathcal{D}_2$)
2) the lowerbound(s) of D' is (are) greater or equal the corresponding lowerbound(s) of D"
3) the upperbound(s) of D' is (are) less or equal the corresponding lowerbound(s) of D"

$D' = D''$ iff $D' \leq D''$ and $D'' \leq D'$

In regard to the *incidence* of domains we shall say D' *is a subregion of* D":

$D' \subset D''$ iff $L(D') \subset L(D'')$

We say that D' and D" *coincide*:

$D' \veeeq D''$ iff $D' \subset D''$ and $D'' \subset D'$

For a good understanding one should observe that each c o n c e i v a b l e domain (element $\in\mathcal{D}$) has essentially three attributes: a type ($\mathcal{D}_1$ or $\mathcal{D}_2$), bounds a n d a *possible region* in L.

Observe that neither $D' \subset D''$ implies $D' \leq D''$, nor $D' \leq D''$ implies $D' \subset D''$, and that neither $D' \veeeq D''$ implies $D' = D''$ (D' and D" may even be incomparible, i.e. of different type), nor $D' = D''$ implies $D' \veeeq D''$ (they may be even in disjunct regions). In inclusion-relations incidence is disregarded, in incidence-relations types and bounds are ignored.

Not until recognizing the essential difference between the two "equalities" $D'=D''$ (mathematically equal domains) and $D'\veebar D''$ (coinciding domains), we arrive at the correct definition of computational equality of domains:

for two domains D' and D'' in $\mathcal{D}$ we have:
$D' \equiv D''$, D' is the same as D''
iff both $D'=D''$ and $D'\veebar D''$

The negative formulation may be more intelligible for our purpose: we consider two domains to be different if they are not both equal and incident.

$\Delta=\Delta_1\cup\Delta_2$ (the set of all concrete domains present in memory at a particular moment) is a very limited and incomplete set. Many feasible domains simply are not there, although the operative subdivision of $\mathbb{L}$ might allow them.

The interesting subset of $\mathcal{D}$ now is a subset between Δ and $\mathcal{D}$: the set ∇ of feasible domains, defined by:

$$\nabla_1 = \{D_1\in\mathcal{D}_1 \mid D_1\subset D_1' \text{ for some } D_1'\in\Delta_1\} \qquad \Delta_1\subset\nabla_1\subset\mathcal{D}_1$$
$$\nabla_2 = \{D_2\in\mathcal{D}_2 \mid D_2\subset D_2' \text{ for some } D_2'\in\Delta_2\} \qquad \Delta_2\subset\nabla_2\subset\mathcal{D}_2$$
$$\nabla = \nabla_1\cup\nabla_2, \text{ hence } \Delta\subset\nabla\subset\mathcal{D}$$

In plain language: at any moment, $\mathbb{L}$ will be subdivided in regions $\mathbb{L}(D)$ described by domains $D\in\Delta$ (the domains realized in memory). These regions may also underlie other domains not yet realized, and these together with Δ form ∇ (are feasible). Observe that there will be many domains in $\mathcal{D}$ not (yet) feasible, but for which we may generate a new region in $\mathbb{L}$.

1.3.2
Concrete operations

In this section we use the term "status quo" to mean the state function Σ: $\mathbb{L}\twoheadrightarrow\mathcal{S}$, together with the subdivision of $\mathbb{L}$ in regions $\mathbb{L}(D)$ at a given moment. Accordingly, we shall distinguish two kinds of alterations in the status quo: due to new assignments in Σ, or due to extension of $\mathbb{L}$ with a new $\mathbb{L}(D)$ - i.e. generation of a new $\mathbb{L}(D)$.

From the given classification of concrete domains, based on "being realized" (i.e. Δ), "being feasible" (i.e. ∇) and "being conceivable" (i.e. $\mathcal{D}$) as discussed in 1.3.1, we now come to the following classification of concrete operations:

<u>Δ-operations</u>:
- they do not alter the status quo,
- Δ and a fortiori ∇ remain the same,
- they compute a truth value, an integer or a scalar.

<u>∇-operations</u>:
- they do not alter the status quo,
- Δ is being extended with a domain $D \in \nabla$ - hence, ∇ remains the same,
- they do not compute anything other than the new D in Δ.

<u>array-assigning operations</u>:
- they alter the status quo in that new assignments of scalars to a certain domain $D \in \Delta$ will be made; hence, they alter the state function $\Sigma: \mathbb{L} \to S$,
- Δ and a fortiori ∇ remain the same,
- they compute the assignment and thereby a new array.

<u>array-generating operations</u> (<u>$\mathcal{D}$-operations</u>):
- they alter the status quo in that $\mathbb{L}$ is being extended with a new $\mathbb{L}(D)$,
- Δ is being extended with a (not yet feasible) domain $D \in \mathcal{D} \backslash \nabla$,
- ∇ will be extended accordingly, i.e. many domains of $\mathcal{D} \backslash \nabla$ go to ∇,
- they compute the newly generated domain.

The array-generating operations are the most drastic because they require new storage; moreover, in the nature of things, they will always go together with (or at least necessitate) an array-assigning operation (one does not reserve storage without doing anything with it). The array-assigning operations are rather drastic in that they alter Σ, i.e. information will be destroyed. The Δ- and ∇-operations are relatively "innocent" (as compared to the other two) because they maintain the status quo.

Typical Δ-operations are <u>lwb</u>, <u>upb</u> and <u>size</u> (computing the bounds and size of a given array-domain) and the predicates <u>fitsin</u> (x <u>fitsin</u> y is the predicate $D_x \leq D_y$), $=$, $\neq$ (they apply to the total arrays!). Another class of Δ-operations form the "sum products" (inner-product, convolution-product, Horner-product), and the simple subscriptions u_i, A_{ij} - they all compute a scalar.

Typical ∇-operations are all definitions of new domains in terms of already realized domains. Examples are diag A (the diagonal of A), A∘i and j∘A (the ith row and jth column of A, for the notation see 2.3.5), the projection of a vector on a subspace etc.

In chapter 5 all the operations in 5.3, 5.4, 5.7, 5.9, 5.10, 5.11 and 5.17 are Δ- or ∇-operations; the operations in 5.5, 5.6, 5.12 and 5.13 are array-assigning operations, and those in 5.1, 5.2, 5.14, 5.15, 5.16 and 5.18 are array-generating.

Characteristic of TORRIX is how it splits the pure mathematical operations +, - and also (though less consistent) × and /, up in array-generating and array-assigning versions: let "□" denote one of +, −, × or /.
Of the expression x□y one normally expects a new value and no side effect on x or y. That is precisely their meaning in TORRIX: the operations "□" are array-generating.
Their (less drastic) array-assigning counterparts will be denoted by "□<" or "□>". The meaning of x□<y is mathematically equivalent to x□y but the result of the operation is assigned to the domain of x (correspondingly x□>y to the domain of y) without intermediate array-generation.
The operation x□<y requires, if both x and y denote arrays, that $D_y \leq D_x$ (y fitsin x). There is an even more powerful operation x□:=y which performs x□<y if $D_y \leq D_x$, but generates a new (better fitting) domain for x if not $D_y \leq D_x$.
Mathematically there is no sensible difference between operations □, □< or □:=. In a computational system they supply in quite different situations the adequate tools and they are very important from the economic point of view.

NB. Equal but partly overlapping domains (i.e. the inclusion situation $D'=D''$ together with the incidence situation $D' \cap D'' \neq \emptyset$) may give problems in the optimization of certain vital operations, specifically of the kind □< (cf. the remarks on 6.0, 6.6.1, 6.6.2, 6.13.5 and 6.13.6). The main difficulty is that the predicate $D' \cap D'' = \emptyset$ is in many cases time consuming and also far from trivial.

2. LANGUAGE AND IMPLEMENTATION

2.1	AIMS AND MEANS	25
2.1.1	Design objectives	25
2.1.2	Implementation language	27
2.1.3	Pros and cons of ALGOL68	29
2.1.4	Routinetexts and separate compilation	30
2.1.5	Optimization	31
2.1.6	Transput and errormessages	32
2.2	THE UNDERLYING SCALAR SYSTEM	34
2.2.1	Scalar and index, TORRIX-REAL	35
2.2.2	Problems of precision	36
2.2.3	Natural, integral, rational	37
2.2.4	Complex scalars	38
2.2.5	Scalar systems with parameters	39
2.2.6	Modops	40
2.2.7	Recursive modops	41
2.3	THE TOTAL ARRAY	43
2.3.1	The two kinds of variability	44
2.3.2	Stack and heap	45
2.3.3	Generating procedures, relation to ALGOL68S	46
2.3.4	Refers	48
2.3.5	Selectors	49
2.3.6	Descriptors	52
2.3.7	Collateral loop-clauses	53

2. LANGUAGE AND IMPLEMENTATION

TORRIX has been, right from the start, a quite serious venture of finding and going certain new ways of software engineering, rather than the umpteenth academic exercise on the construction of a vector-matrix package. Accordingly, the problems of reliability, safety, consistency, adequacy, completeness, make-up of the users interface and of efficiency in time and space, have been scrutinized. In a few conflicting situations we let our priorities correspond to more or less that order - the highest priority being reliability, the lowest efficiency in space (cf.2.1.5). However, there have been - surprisingly enough - not many conflicts, and it may be that keeping away from them was our intuitive overall guiding principle. May be also that they will not really conflict, provided that you treat them well! For the results of our deliberations the reader is referred to the following chapters - the present one is focussed on the deliberations rather than on the outcomes.

We also discuss here, in some detail, the principal (i.e. non-technical) aspects of the implementation and of the choice of the implementation language. The two, of course, are related. By their very nature, the objects and operations under consideration (see chapter 1) require a high level programming language which, nevertheless, will delimit the implementation to some extent. At each point where a limitation really hurts and the kind and cause of the pain could be determined with some precision, we have in fact exposed a shortcoming of the language - in many cases an imperfection of its design. Therefore, this chapter can also be read as a report on how the programming language in question did or did not sustain a rather exacting test.

2.1 AIMS AND MEANS

This section is to account for the design objectives and the more general implementation criteria. What we are implementing has been described mathematically in chapter 1 - a "TORRIX-system" T. Here we discuss where we aim at with this implementation and by what means we shall proceed.

Chapter 1 immediately leads to a kind of exclusion principle, preceding the actual design principles:

- TORRIX should not contain any object or operation which is alien to the intrinsic features of linear spaces (1.1.2).

For an example we consider the operation of ordering (sorting) an array. Without any doubt this is an important candidate - nevertheless we did not admit it. For one reason, ordering is not a linear operation; but, apart from that, it is even impossible to give it an appropriate (be it studied or even far-fetched) interpretation in the context of linear spaces - it is truly alien. We can sort of prove this: following chapter 1 (see 1.2.2) the total-array concept is in full agreement with the axioms of linear spaces - now, imagine the total-array of say (-1,+1), sort it in non-decreasing (!) order and observe how a simple 2-dimensional subspace explodes into the total space of (-1,0,0,0,-------,0,0,0,+1).

This, of course, does not imply that a TORRIX-user is not entitled to sort an array if he feels like doing so - especially to sort an *index* (see 3.2.2) for which he may have good reasons. He should, however, realize that this then is a pure administrative action - very much like counting iterations or something - and as such it has nothing to do with T.

2.1.1
Design objectives

From the abstract approach discussed in chapter 1, the following general desirabilities emanate for practical application:

I Free choice of the underlying scalar system S.

That means the possibility of applying (the elements of) S without any irrelevant specification - i.e. it is expected that the S-operations are available without a priori specification of how they work. Or, to put it differently, it is required that we can write complete programs in T (over some S) which, as such, are valid for different choices of S - be it R in any precision, or Q, or Z_p, or whatever may be appropriate. In particular the possibility of exploring different systems (different precisions) for R, may be of great practical value.

II Vectors and their linear transformations (matrices) act
as autonomous (i.e. non-derived) objects.

Hence, our view of a vector space is coordinate-free on principle. This implies that, wherever we can formulate a process without reference to a concrete array or its scalar elements, we shall be able to program it that way. In other words: although we all know how vectors and matrices are so to say cooked in the kitchen - for consumption we definitely prefer the menu as it is dished up by the rules 1, 2 and 3 in 1.1.2 and 4 in 1.1.5.

III Full independence of the dimension of the particular
vector (sub)space(s) in which we operate.

For this requirement, of course, we invented total-arrays and operations on them, obeying and respecting the laws of linear spaces. The very scope of this claim in the practice of programming is, that vector spaces of different and even at runtime varying dimension can be manipulated without precaution.

The three principles together blueprint TORRIX as a system in which scalars (in any realization of the formal concept) and their vectors and matrices can be combined in such a manner that they not only fully satisfy the practical requirements of applied (numerical) modern mathematics, but also do justice to abstraction in both type and dimension.

A fourth principle is on the implementation itself, expressing that all of I, II and III be done with negligible extra costs (if any) as long as the user stays within the limitations of the more traditional vector/matrix systems - and preferably still even if he goes (not too far) beyond:

IV The programmer does not pay for those particular features he does not use.

We applied IV specifically for the critical area of storage-allocation and memory-access, for which we refer to section 2.3.2.

2.1.2
Implementation language

There were essentially three options for the implementation of TORRIX:

1) Define it as an independent and autonomous programming language; write a compiler for it and an adequate running system.

2) Take a suitable existing programming language and extend it for the purpose; extend the compiler and the running system accordingly.

3) Take a suitable existing programming language and write a TORRIX-library within it (without doing something to the language itself).

The most satisfying of the three, undoubtedly, is the first. It is also the only way by which we do not have to compromise. And it would enrich the world with one of the next sevenhundredandsomany new programming languages. We have considered it, but with a minimum of enthusiasm - in spite of the challenge and temptation in being free in a private world without constraint.

The second alternative has its charms too - a substantial part of the definitional work and compilerconstruction has in fact been done. A good example of how to proceed can be found in {26} which describes a language VECTRAN, extending FORTRAN. This language is also on vectors and matrices, so it is a good instance. However, this VECTRAN does (and can do) nothing about our main objectives I and III - though it does a little bit about II (IV is not applicable). Moreover, regrettably but inevitably, it also displays all the well-known shortcomings of its parent language.

The main problem, of course, is I: the requirement of being enabled to program in terms of an abstract data-type S. Clearly, in the third alternative we are also faced with this problem, having even less prospect of finding a solution. So the question actually is: what language allows for the definition of abstract data-types and of operations for them?

It happens that ALGOL68 comes pretty close to that. 'Mode' is just another word for "type" and the ALGOL68 'mode-declaration' is in fact a definition of a new data-type in terms of already knowns. For new modes we can also declare new operations, even using the appropriate symbols such as "+", "-", "×" and "/". On top of that, ALGOL68 has also a fairly good set of built-in features for manipulating arrays of different sizes, even pretty close to what we need for III (we give a survey in 3.1.3).

One of the benefits of a good language is the aid, and even the inspiration, it may give in getting ideas and in designing systems. From the beginning, ALGOL68 has been our main vehicle for the development of TORRIX - the traces can be found in {14}, {23}, {20} and {24}. It would not be easy to determine what of TORRIX comes more or less directly from ALGOL68 and what would have been invented anyhow.

However, it is a remarkable fact that in TORRIX, as presented here, our three seemingly rather disjunct alternatives become less exclusive:

-1) The declarations in chapters 4 and 6 describe clearly and completely the data-structures and operations of a system which presents itself apparently as a new programming language. It would become an entirely independent and autonomous one, if we added the appropriate control-structures - this can not be done within ALGOL68, from which we now had to borrow them.

-2) TORRIX is a true extension of (the standard-prelude of) ALGOL68. It demonstrates to what degree ALGOL68 is an extensible language: for types and operators yes, for control-structures no.

-3) TORRIX is, technically, a true 'library-prelude' within ALGOL68 - i.e. it has been implemented by no other means than ALGOL68 (for a few in this context negligible exceptions see 2.1.4, 5.7 and 6.7).

In the sequel the word TORRIX will be used to denote the vector-matrix language as implemented in ALGOL68 plus, occasionally, those dreamt of things we did (or could) not express in ALGOL68. If we want to refer specifically to the TORRIX/ALGOL68 'library-prelude' proper, we shall use the word TORRIX68.

2.1.3
Pros and cons of ALGOL68

The pros and cons of ALGOL68 for the TORRIX implementation appear in detail from the remainder of this chapter. At this place we discuss a few generalities.

The language has been constructed by the principle of "orthogonal design", a somewhat peculiar term meaning that "the number of independent primitive concepts has been minimized", and that "these concepts have been applied 'orthogonally' in order to maximize the expressive power of the language while trying to avoid deleterious superfluities" (we quoted the Report, see 0.1.2 in {36}). Apparently, 'orthogonality' means so much as 'in all possible combinations, with a choice of primitive concepts that makes all combinations possible'. The orthogonality of the language is commendable in the realization of the mode concept and also in most of its control structures. Nevertheless, we ran up against a few imperfections, some of them were a nuisance.

We have been hampered by both insufficient primitives and inadequate combination of them. In a sense it is ironic that a language which is quite a show of consistent construction, failed precisely in those small corners where it was just a tiny little bit not consistent. We can now say that its orthogonal design gave ALGOL68 an impressive power - with, orthogonally, likewise remarkable weak little spots.

Our overall conclusion is, accordingly, that ALGOL68 was for our purpose certainly not the huge, overdoing language for which some people still seem to take it. Quite to the contrary: where it failed, it was in fact underdoing - in its own spirit.

On the other hand, we did not need all the independent primitive concepts. Significantly, we could not use *flex* (see 2.3). More to be expected we did not need *par* and *sema*. In the present volume we could also do without *union*. We may need it for some of the more demanding data-structures in TORRIX-SPARSE.

We stayed also away from procedural data-structures (i.e. data-structures which are given by *proc*s, see {19}) for the obvious reason that ALGOL68 disallows (by scope-restrictions) *proc*s defining *proc*s. This, undoubtedly, was a painful limitation, although we did not deeply examine the possibilities we missed, (see, however, 3.2.5 and 5.5).

As to the many pros, we may refer to the routinetexts in chapter 6. If they do not speak for TORRIX, the least they do is speak for ALGOL68.

A final remark on structured programming. After all discussions, controversies and things "considered harmful" - including sometimes programming itself -, we do not know anymore what it is. Our approach has been a conscientious mathematical analysis of the subject matter (chapter 1), resulting in a bottom-up synthesis of the objects and operations needed (chapter 6). Here the routinetexts may speak for TORRIX, the least they do is reveal its structure. Chapters 2 to 5 may reveal how we got from 1 until 6.

2.1.4
Routinetexts and separate compilation

All the routinetexts of TORRIX-BASIS in chapters 4 and 6, released by this publication, have been carefully tested on the CYBER/ALGOL68 compiler version 1.1 at the University of Utrecht. For particularities on this compiler we refer to the users manual {03}. The only difference between the declarations as presented here, and those in the sourcetext listing from punched cards, is their representation.

In this publication we follow the representation style of {36}, but we have used underlining to indicate boldface typefont. In the original sourcetexts we adopted the open-and-close-apostrophe stropping convention for bold characters (see {37.3}). The taboo-mark "†" (see 2.2) corresponds to a specific combination of punched characters, available on the CYBER/ALGOL68 compiler for the denotation of characters not denotable by the unpriviliged user (see also 10.1.3 Step 2 in {36} on ?).

In 6.7 a few routinetexts (marked with a *) have been defined through informal statements between open-and-close-pseudocomment-symbols "*C*". These actions cannot be defined in ALGOL68 proper. However, on each full implementation of the language, it must be possible (and by quite simple means) to incorporate these particular actions in the compiled code. The CYBER/ALGOL68 compiler provides for such insertions through 'pragmats' in an intermediate code (for 'pragmat' see 9.2 in {36}). Our pseudocomments correspond to such pragmats. We leave it to the ALGOL68 exegetes to decide wether we remained inside ALGOL68, or made just a few tiny little steps outside (see also 3.2.5, 5.7 and 2.3.6).

All other aberrations from the original sourcetext must be due to typing errors which then escaped several scrupulous iterations for correction.

Any main program written for TORRIX-BASIS can be compiled, being linked to a precompiled binary TORRIX-file for each choice of *scal* (see 2.2). The loader will then select precisely those routines which were, directly or indirectly, required by the main program so that the final object program will not take more memory space than it actually needs.

These facts refer to the 1.1 version of the compiler. Shortly before we were going to press, the CYBER/ALGOL68 compiler version 1.2 became available. This new version includes many improvements in both compilation and generated code (optimized variables, improved descriptors etc.).

2.1.5 Optimization

The routinetexts as such are optimal so far as could be expressed in ALGOL68. We respected, however, the intrinsic bottom-up structure of the system. The given scalar operations form the basis for the scalar to vector and the vector to vector operations (sums, products etc.), these in their turn are the primitives for the matrix to vector operations, in terms of which then finally the matrix to matrix operations have been declared.

They all come down to operations on concrete arrays, which is why the *max* and *min* operators on lower- and upperbounds of arrays play a key role - they determine the ranges of the do-loops which carry the (optimizing!) total-array strategy into effect. This bottom-up structure is quite obvious from the order in which we present the routinetexts (see 4, 5 and 6). Therefore, it cannot be difficult to find out, for a specific implementation, which routines will be the first candidates for optimization. Most likely, the outcome will be quite similar to the situation with the CYBER-system:

- The interplay of the standard operators *lwb* and *upb* with *max* and *min*. They cooperate closely with the concrete-array descriptors, and several shortcuts must be possible, specifically in constructions such as "*from lwb u max lwb v to upb u min upb v*". Though they are of a purely administrative nature, these operators occur so frequently in the text that they deserve optimization not only for saving CPU time, but also for saving generated code (see also 2.3.7 for this matter).
- The operators *fitsin*, *into*, *copy*, *span* and *inspan* serve almost everywhere in TORRIX-BASIS and, depending on the implementation, various optimizations will be possible and necessary.
- All hidden operators (marked with "†", see 6.0) as also the basic operations *=:=*, × (which is the same as <>) and ><, on the concrete array level.

These optimizations alone will accomplish an expectedly 40% overall improvement in performance on the CYBER/ALGOL68 system. Important further optimization can be done to the do-loops so as we had to formulate them (this is the subject matter of 2.3.7).

Though they are dispensable in TORRIX-BASIS, obvious optimizations will be possible for:

- The operators *?* and *//* and their "shadow-modes" *pair* and *trimmer*. It must be easy to implement these total counterparts of the standard ALGOL68 array selection and slicing actions, in such a manner that they have precisely the same performance (see also 2.3.5).

It may heavily depend on the specific ALGOL68 implementation how to proceed in optimizing TORRIX-BASIS. The best way to attack the problem on the CYBER/ALGOL68 compiler, will be to replace - in a well-planned order - one routine after the other by "hand coded" 'pragmats' (just as we did to the * routines in 6.7).

2.1.6
Transput and errormessages

The ALGOL68 standard transput is, apart from imperfections in its definition, more than enough powerful to cater for all demands in the matter of the input and output of concrete arrays. However, the typical TORRIX interplay of an abstract type *scal* and its total-arrays, make it advisable to have a few specific transput-facilities so that also the transput statements in a program become invariant over the choice of *scal*.

Apparently we need some kind of format-parameter(s) for *scal* and a possibility to transput the descriptors of concrete arrays together with the arrays themselves, and to control how many *scal*s on a line and how many matrix rows (or columns) on a page we want to output. Because of a few further requirements stemming from the more advanced TORRIX-applications, we postpone further discussion until the second volume.

Part of the transput are the errormessages (see 4.2 and 6.0.9). We distinguish two kinds of special events:

1) Fatal errors.
   They always lead to a program abort. The accompanying message, reporting a fatal error is, necessarily, the last piece of output.

2) Non-fatal errors or -events.
   They do not impede the execution of the program; their only side-effect is a "warning" sent to the errorfile, reporting the event.

Essentially, a warning may report an unintended, but not fatal error or, more often, an intended event worth, however, to be explicitly mentioned. A fatal error, most likely, will never be intended; anyhow it terminates the execution of the program in which it occurs.

The error file is not necessarily connected to the same channel as the standard output file - it is, in all cases, an other file. The physical output of the errorfile is suppressible.

The TORRIX message system, as it has been declared (in 4.2) and applied (in 4.3 and 6), is optional. All warnings may be drastically simplified or even left out; all fatal error messages may be reduced to a straightforward program abort.

Our proposal is a kind of optimal interpretation of the "undefined" in the ALGOL68 Report (see 1.1.4.3 in {36}), as applied to TORRIX68.

2.2 THE UNDERLYING SCALAR SYSTEM

The ALGOL68 facilities for the fulfilment of objective I are quite reasonable, though not ideal in every respect. Where ALGOL68 is the only available language with such facilities, one should not complain too much.

Operators in ALGOL68 can be declared as generic procedures - i.e. it depends on the mode of the operand(s), which routine will be selected for a given operator symbol. For example, $a+b$ means integer addition if the mode of a and b is *int*, but textual concatenation if their mode is *string*. On parsing, the compiler will select from all available routines for "+" precisely that (presumably unique) one declared for the required mode(s) - e.g. an *op(int,int)int* if both a and b are of *int* mode, an *op(int,real)real* for an *int* and a *real*, etc. and in TORRIX68 also an *op(scal,scal)scal*, an *op(vec,vec)vec* or an *op(mat,mat)mat* if a and b are *scal*s, or *vec*s, or *mat*s.
It is, however, required that all the modes and routinetexts be known at compile time. Consequently, it is not possible to (pre)compile a TORRIX system independently of its underlying scalar system. Of course, the scalar system can be precompiled before we go into compiling a TORRIX system, but there will be as many TORRIXes as there are scalar systems, and they will all be different. As compared to complete independence (i.e. one, general, precompiled TORRIX can be linked to any scalar system) this is a limitation, but not a serious one. We are inconvenienced by a certain practical inflexibility, no more.

A scalar system constitutes a mode/operator package, usually smaller but quite similar to the TORRIX system it underlies. In 2.2.7 we shall come back at a certain implication of this observation.

An ALGOL68 mode/operator package is an entirely unstructured set of declarations. The only possibility of putting it in some order is by writing the declarations in some appropriate succession (that is what we tried to do in chapter 6). There are, however, neither adequate tools for distinguishing different layers of relevance (not even for the protection of specific privileged information), nor parameters for the package as a whole (or for parts of it). We shall come back to this in 2.2.6.

Of course there are certain ways out. For the protection of routines and modes which are dangerous for the user (i.e. for "layering" to some extent) we have a rather ugly solution by which we remain within ALGOL68: to redeclare their identifiers or indicants after an 'open-symbol' (shoved between the package and its next "layer"). We followed a more practical solution using a 27th letter "†" (pronounced "taboo"), which we could do by a special facility of the CYBER/ALGOL68 compiler (see also 2.1.5). For the simulation of package-"parameters" we can, of course, use globals (the almost prehistoric solution).

We shall discuss in this section a number of small difficulties with the set-up of scalar systems. They can all be solved, and the solutions demonstrate where the language missed a point.

2.2.1
Scalar and index, TORRIX-REAL

An essential requirement for our scalar systems is, that they contain the integral domain Z as a subsystem: $Z \subset S$ for all S (cf. 1.1.1 and 1.2.1). For TORRIX68 this comes down to *int* ⊂ *scal*. In a completely puristic implementation we would have a separate mode index for subscripting the arrays and all their derived modes. The set index is finite; the set *int*, essentially, is infinite (though usually implemented as a large finite set). Accordingly we would then require index ⊂ *int* ⊂ *scal*.

In ALGOL68 we have index = *int*. In TORRIX68 we confined index to the interval [-m:m] where m=*maxdex* (see 3.2.1). Now, for *mode scal = real* (the normal case so to say), everything works fine: we have a kind of automatic widening from *int* (index) to *real*. "Widening" means that in all real (complex) operations, an integral operand will be treated as (widened to) a real (complex) operand.

Unfortunately, there is no automatic widening in ALGOL68 from *int* to any *scal* other than *real* or *compl*, not even to *long real*. This is a shortcoming of the language: an index in *long real* arithmetic must now be treated differently from an index in *real* arithmetic. We are allowed to write *i*×*u*[*i*] with *scal* = *real*, but not with *scal* = *long real*. Even the *int*-denotations (notably *0* and *1*) cannot be used in combination with any non-*real* (non-*compl*) operand. Observe that our complaint applies to index rather than to *int* (the parity of *long int* and *long real* seems to be all right, but it was a mistake to ignore the distinct position of an index in this matter).

The only decent solution for TORRIX68 is to declare an operator *widen* which performs, for all choices for *scal*, the "widening" from *int* to that *scal*. In order to take the burden away from the programmer, we systematically defined, next to all operations for *scal* with *vec* or *mat*, a version for *int* with *vec* or *mat*. That accounts for quite a number of operation-declarations in chapter 6. For *scal* = *real* we can simplify the system; see below the remark on TORRIX-REAL.

However, ALGOL68 does not allow redeclaration of := (assignation). Consequently, one trap remains: $u[i]{:=}i$ is correct only for *scal* = *real* (*scal* = *compl*). For this we have no other remedy than an advice: if you cannot avoid it (compare *into* in 5.5, see also 3.2.5), be wise and write always $u[i]{:=}$ *widen* i in order to keep your program independent of the particular choice for *scal*.

The proper place for the declaration of *widen*, naturally, would be the mode/operator package for *scal* (see also volume II). Consequently, some small measures should be taken in linking TORRIX to a particular (home-made) *scal*-package.

It will be clear that the (probably most common) choice for *scal* = *real* is now being saddled with all kinds of provisions for other horses. This is why we recommend to maintain a system TORRIX-REAL in which *widen* is declared to be a dummy (it may be used in the main program!), though it has no applied occurrences in that system.

Apart from *widen* there are also other, rather important, simplifications and shortcuts possible in TORRIX-REAL (the multiplication is commutative, to take an instance). At this point the reader should be well aware of the fact that TORRIX-BASIS, as we present it in chapter 6, is the implementation of the most general case. In TORRIX-REAL, as also in most other versions, many details (dependent on the choice for *scal*) can be simplified or even left out.

2.2.2
Problems of precision

The real system R can only be represented as a noncontinuous and finite approximating subset $R' \subset R$ (cf. 1.2.1). This can be done in different precisions. To that purpose we have in ALGOL68 a whole procession of *real* lengths: *L real* can be ---- *short short real*, *short real*, *real*, *long real*, *long long real*, ----, etc. This may be the only place where ALGOL68 is really overdoing. Anyhow, we think that the TORRIX way of treating precision is both more practical and more general.

Our point of departure is that it will very rarely, if ever, occur that more than two real precisions are needed in one execution of the same program. Hence, what we need is at most one long scal in addition to *scal* - provided that we can choose different precisions for *scal*. ALGOL68 should have done the same thing, maintaining one extra (double) precision "*long real*", and playing the precision choice of *real* over an execution-parameter. We cannot syntactically define "*long scal*" in ALGOL68, so we called it "*scalon*".

This approach also prepares the way for choosing for *scalon* something different from a real in greater precision. For example: *mode scalon = struct(scal lower,middle,upper)* for interval arithmetic as an addition to the normal scalar arithmetic. In volume II we shall come back on *scalon* and its possibilities.

2.2.3 Natural, integral, rational

Where TORRIX can be linked (i.e. compiled together) with any scalar system, not even necessary an algebraic field (cf. 1.1.2), some particular choices for *scal* may be interesting:

mode scal = natural

Here *natural* is a supposed implementation of N as it should be, i.e. without overflow-limits. Ways to implement some such *natural* are well known, anyhow not difficult to invent. For example:

mode natural = struct(int digits, ref natural overflow)

The *digits*-field contains a (machine- or implementation-dependent) number of (binary or decimal) digits. The *overflow*-field refers to the next item in the *natural* chain containing the possible overflow. The *natural* operations "+" and "×" are easy to implement (and with a reasonable efficiency);

"*over*" (integral division) and "*mod*" (the remainder) may give more difficulties in case the divisor has a non-*nil* overflow. For the ordering relations "<", "<=", ">" and ">=", as also for "=" and "/=", we declared an operator "-" which is, however, a partial operator in N. Applications for TORRIX-NATURAL may be found in the theory of numbers.

mode scal = integral

Here *integral* extends N to an overflow-free Z. For example:

mode integral = struct(int signtail, ref natural overflow)

The *signtail*-field contains the least significant digits of the *integral*. The "+", "-", "×", "*over*", "*mod*", "<", "<=", ">", ">=", "=", and "/=" are extended into Z. Z (*integral*) is a true algebraic ring and consequently TORRIX-INTEGRAL will be a true algebraic module. Observe that *int* (index) needs a (rather trivial) *widen* operation for becoming an *integral*.

mode rational = struct(integral numer, natural denom)

Of course *rational* will be a true implementation of Q and thus will be an unrestricted field containing also "/" and not any more "*over*" and "*mod*". The *rational* arithmetic should contain all combinations with *integral* as a left- or a right operand and also a *widen*ing from *int* (index) to *rational*. Observe that all computations in *rational*, and consequently in TORRIX-RATIONAL, work with absolute precision.

A simpler, but limited, implementation of Q rests on:

mode rat = struct(real val, int num, den)

for which we refer to volume II.

2.2.4 Complex scalars

Quite often the necessity to operate with complex vectors and matrices proceeds from the phenomenon of "complexification" in operations on real vectors and matrices, which is why we want to maintain the real vectors and matrices next to the complex ones. So, instead of declaring a *mode scal = compl* (which remains possible after some accommodation of TORRIX-BASIS, see volume II), we prefer to extend TORRIX-BASIS with a TORRIX-COMPLEX. Therefore we shall have the mode-declarations:

mode coscal = *struct(scal re,im);*
mode coscalon = *struct(scalon re,im)*

For their use we refer to volume II. Here we only mention the very nice way in which the quite subtle ALGOL68 mode-equivalencing works for us. For all choices *mode scal* = *L real* we get, automatically (and at compile time), *coscal* = *L compl* and *coscalon* = *long L compl* (provided that *scalon* = *L real*).

For a less nice aspect of the ALGOL68 *compl* we refer to section 2.3.4.

2.2.5

Scalar systems with parameters

Algebra is an inexhaustible source of fields, rings and other systems with all kinds of nice and nasty properties. Many of them may, in the proper axiomatic frame (cf. 1.1.2), underlie a vector/matrix system. Here we mention two such scalar systems for no other reason than the typical problem of their implementation in ALGOL68.

Z_n is the finite system of integers modulo n; if p is prime, then Z_p is a field. $Q(\sqrt{d})$ with $d \in Q$ non-square, is the quadratic field; its general member is of the form $\sigma = r + s\sqrt{d}$ with $r,s \in Q$. The operations in both systems obey the operations in the system from which it is derived - Z for Z_n and Q for $Q(\sqrt{d})$. However, a system-parameter now plays a key role: in Z_n we must reduce all operation-results modulo n, and in Q we have to split them into a pure rational part r and a factor s of $\sqrt{d}$ (of course $\sqrt{d} \times \sqrt{d}$ comes down to $d \in Q$).

Consequently, we have to face the problem of a "package-parameter". What we need is something like:

mode modulo(int n) = *struct(int m)*
mode quadr (rational d) = *struct(rational r,s)*

The type font indicates what we can do in ALGOL68. Having such a formal package-parameter would imply the possibility of actualizing it. This might then be done by parametrizing precisely that piece of program (presumedly an 'enclosed-clause') in which it should have a particular value, say n=37, d=3/2. It should then also be possible to specify which parameter is to be set to the required value because more than one parameter in different packages (depending on different modes) may play a role. So that we would get something like:

> (# piece of program using *modulo* #)(*n* of *modulo* = *37*)
> (# piece of program using *quadr* #)(*d* of *quadr* = *3*/2)

possibly even:

> (# piece of program using both *modulo* and *quadr* #)
> (*n* of *modulo* = *37*, *d* of *quadr* = *3*/2)

Of course, we still can attain our goal by playing the parameter via a global variable, simply assigning *n:=37* or *d:= 3* *div* *2* (for *div* see 7.1.3.7). This solution, however, has its limitations, both of organizational and of practical nature.

2.2.6 Modops

The previous considerations lead quite naturally to requirements for future languages, in particular for the equipment of mode/operator packages - or "modops" for short.

Modops have been (and still are) considered in various contexts. They are known under different names, such as "preludes" (the official ALGOL68 name), "classes", "modules", "clusters" etc. For their discussion we refer to the litterature, e.g. {08}, {12}, {18}, {27}, {28}, {29}, {30}, {33}, {36} and notably {15} and {31} - the list is far from complete. We shall go no further than briefly summarizing a few wishes arising directly from the TORRIX project. The subject on its own is very interesting.

1) Modops should have good provisions for the definition of different "layers of relevance". In each layer we should be enabled to decide which entities (modes, values, routines, operators, identifiers etc.) may be known to the upper layer (and finally to the outside world), and which ones will be local to the layer.

Of course, a layer is to some extent a generalization of a "block". The problems of scope, however, are considerably less simple in them and, apart from that, open for various improvements anyhow (see also {06} and {12}).

2) Modops should be separately compilable.

We mean this in the rather strong sense that, even if different modops need each other, as TORRIX needs a modop for its *scal*, they can be coupled after their (separate) compilation. This is a matter of both language design and implementation.

3) Modops should have good provisions for parameters - not only values, but preferably also modes and other entities.

Hence, modops appear to become something like "large, complicated procedures". You have to put information (actual values, modes, operators etc.) into them and out of them come new datatypes defined in terms of all the allowable operations on them (of which many may be founded in deeper layers).

Observe how in the modop named TORRIX (chapters 4 and 6) the mode *scal* is used as a kind of formal mode for which operations "+", "-", "×" etc. are supposed to exist with certain properties. A mode-declaration *mode scal = real* or *mode scal = rational* etc. couples this modop TORRIX to the modop for *real* (i.e. part of the standard-prelude of ALGOL68) or *rational* (a home-made modop). In fact the general modop TORRIX(*scal*) has been actualized to TORRIX(*real*) or to TORRIX(*rational*) respectively.
Correspondingly, we would get through *mode scal = quadr*(rational *d*) the actual modop TORRIX(*quadr*(rational d)) which then is a coupling of three modops: TORRIX, *quadr* and *rational*. Observe that TORRIX is a modop defining more than one mode (*vec, mat, index* and those discussed in the second volume).

Potentially all these things can be done in ALGOL68 - that is where this book is about. However, our instruments are still rather primitive; we need better tools.

2.2.7
Recursive modops

A quite interesting phenomenon shows up in observing that the scalar system S, underlying T, may be (at least partially) a vector-system. This is already, in a certain minimal sense, the case if we take for S the complex field C (additively a 2-dim. system), and more so if S is the Hamiltonian skew (non-commutative) quaternion field H which is a 4-dimensional vector space over R.

Without implying anything concerning its mathematical relevance, a pertinent example is to take the polynomial ring P for S - i.e. to consider the vector-module V over P - which itself is a vector space over, say, Q. Let us first see how we can achieve this in TORRIX68:

We first precompile P over Q with *mode scal = rational* (or *mode scal = rat*). Now the *vec*s of this modop for P(Q) represent our polynomials. We do not need the *mat*s, so we can leave them out, and we take the convolution product (5.18) as the polynomial product. Accordingly we then redeclare: *mode poly = vec*.

After that we set *mode scal = poly* and we compile TORRIX again, but now together with the precompiled P(Q), in order to get V(P(Q)). Now look what happened: we compiled TORRIX twice (apart from leaving out something, and a minor variation in one operator-definition in the deepest version P). That is: we got two binary files (which may or may not be in principle identical, depending on the implementation). Both came from the same sourcetext.

Then we start to realize that such things occur quite frequently in generic procedures (cf. introduction 2.2) - that for different operand-modes the same sourcetext can be used with consistent substitution of the modes concerned. The authors of the ALGOL68 report were already more or less aware of this phenomenon where they introduced, for the case of brevity, the generalizing pseudo-operator symbols *P*, *Q*, *R* and *E* in their modop named "standard-prelude" (cf. section 10.1.3, Step 1 in {36}).

The crux of the matter, of course, is that we then often go in recursion over the modes. Consequently, although we do not yet clearly see how to implement that kind of recursion (orthogonally, and with all its implications), we nevertheless come to the following requirement:

4) It should be possible for a modop to use itself recursively over its mode-parameter(s).

The conceptual organization of modern algebra - specifically visible in category theory (cf. {16}) - is a strong indication that recursive modops may become quite powerful instruments for abstract type manipulation. Even apart from the strict mathematical context, such a feature may become important - one might think, for example, of the relational (algebraic) approach to databases.

2.3 THE TOTAL ARRAY

The ALGOL68 facilities for the implementation of the total array idea (the way by which we realize design objective III) are quite good. At the most essential point, namely the facility of manipulating concrete arrays of different and possibly also varying size, our technical requirements are even met upto the full hundred percent.

Not until carrying the idea through its farther reaching consequences, we begin to encounter difficulties. Some of these are quite unnecessary (e.g. 2.3.6), others suggest better (future) language features (2.3.4 and 2.3.5). The most important of them, undoubtedly, is the (absent) collateral loop-clause, discussed in 2.3.7. For a different exposé, less depending on TORRIX, we refer to {21} and {22} as also to the litterature given in 2.2.6.

Surprisingly enough, we could not apply the ALGOL68 basic concept of a "flexible name" (i.e. an object of the mode *ref flex*[ ]*amode* or *ref flex*[,]*amode* etc.) which was invented for the purpose (cf. section 2.1.3.4.f in {36}). The only, but sufficient reason is that a slice of such a *ref-flex*ible multiple value (see 3.1.3 for "slicing") cannot be passed as an actual parameter to any routine in which the formal parameter is *ref-flex* (cf. sections 2.1.3.6.b and c in {36} for the precise formulation of the constraint on so-called "transient names"). This prohibition is fatal for our purpose in which slicing plays an essential role (see also 1.3.1 and 2.3.5 below).

It is still more surprising that nevertheless we can do, all we want, without *flex*. TORRIX demonstrates clearly the superfluity of the entire concept. All wishes concerning the flexibility of array size (the provision by which arrays may "breathe"), can be accomplished without using *flex*.

The point is that the concrete bounds in an ALGOL68 multiple value become "formal" after a *ref*, i.e. the mode *ref array* does not depend on the actual bounds of the particular *array*. This implies that a reference to such an object (i.e. a *ref ref array*) will accept, in all syntactic positions (notably in the left hand side of an assignment), all bounds of the *array* in the depth. We call such a *ref ref array* a depth-reference - it can replace the *ref flex array* in every respect and do even more. For a more complete discussion see {21} and {22}.

At one point the TORRIX extension of a concrete array to a total one breaks with the ALGOL68 view on a "multiple value". The out-of-bounds elements are undefined in ALGOL68 whereas in TORRIX they are specifically defined to be zero. Clearly nobody is to blame for that disagreement. We extended a partial function to a total one and we did so for a particular application area - for other applications one might need other extensions if any. A universal programming language should not decide in such matters. Consequently we have to define our own selectors and slicers for total arrays if we need them. This is where 2.3.5 is about.

2.3.1

The two kinds of variability

It follows almost directly from section 1.3.1 that we can have in principle two kinds of variability for vectors and matrices. There is a variability caused by the supersedure of old values by new ones within the existing concrete domain, and there is a variability of (the location of) the domain itself - not necessarily leading to a different vector or matrix (e.g. same vector, different concrete domain). The first kind is the common and well-known variability due to assignment, the second kind can make the concrete arrays breathe (the bounds become variable). The ALGOL68 fundamental concept of *ref amode* enables the implementer to make a clear distinction between both kinds of variability.

In order to maintain them and to keep them apart we shall consistently speak of two levels of *ref*erence:

- the direct level of vectors and matrices with fixed bounds:

  mode vec = *ref* []*scal*   *mode covec* = *ref* []*coscal*
  mode mat = *ref*[,]*scal*   *mode comat* = *ref*[,]*coscal*

- the indirect level of vector- and matrix-variables where the bounds become flexible; the modes of these variables are:

  ref vec = *ref ref* []*scal*   *ref covec* = *ref ref* []*coscal*
  ref mat = *ref ref*[,]*scal*   *ref comat* = *ref ref*[,]*coscal*

We call the indirect level (two *ref*s) "level2", the direct level (one *ref*) "level1", and the lowest level (no *ref*) "level0" - that is the plain concrete *array* level. A level0-object can only exist with concrete (known) bounds; a level1-object will normally refer to a level0-object (unless it has not been initialized), but it has no preference for any bounds (not even for a lower bound *1*); a level2-object does not directly refer to an *array* and does not even require its existence, it deals exclusively with the level1-object (*ref array*) it refers to.

In assigning to a level1-object (*vec, covec, mat* or *comat*) old values in the given level0- (i.e. concrete array) domain are superseded by new values; nothing happens to the domains (which remain what and where they are). In assigning to a level2-object (*ref vec, ref covec, ref mat* or *ref comat*) the lefthand side will be made to refer to another level1-object (*vec, covec, mat* or *comat*); the values in the concrete domains involved, have nothing to do with the happening.

For further details on the TORRIX levels and their role in assignations, we refer to 3.1.4, 3.1.5 and all of 3.3.

A very specific kind of variability is caused by "trimming" and/or shifting the concrete domain (see 3.1.3 and also 3.2.5). Clearly a new level1-object comes about: a shift and/or a projection of the original level1-object. The excellent ALGOL68 slicing feature makes that we can perform this specific kind of domain-variability on the direct level of vectors and matrices (i.e. on level1). For details we refer to 3.2.5.

2.3.2
Stack and heap

Where our basic modes (*vec, covec, mat* and *comat*) are *ref*-modes, it is syntactically required that concrete *array*s for use outside a routine-text are generated by means of a *heap*-generator. A *heap*-generator reserves storage in a memory-region, termed the "heap", in which garbage-collection techniques may be used for storage retrieval. Now, as long as we stay at level1 (i.e. at the direct level of *vec, covec, mat* and *comat*), there will never be any storage to retrieve because the information lastly generated is also the first to get rid of. The reason is, that neonate arrays will either be "ascribed" (cf. 3.1.5) to a local identifier, or only exist as an intermediate result. In other words: at level1 the heap works as a last-in-first-out memory and could equally well be implemented on top of the working stack. Not until level2 we may actually need a garbage collector for the holes that may then come about in the reserved storage.

We thus have the following situation:

- If we confine ourselves strictly to the level1-operations, we do not actually need a garbage collector and our "heap" may be combined with the working stack.
- If we apply level2-operations (see 3.3 for their possibilities), we presuppose an efficient garbage collector.
- For syntactic reasons we have to generate all our concrete arrays by means of a *heap*-generator, even when we confine our TORRIX program to level1-operations.

The question of garbage and, eventually, of how to collect it, thus depends on the level on which we operate. The question may be important for the efficiency of our program as also for whether we stay within a certain subset or not (see next section 2.3.3). This is why we demarcated carefully a border between the two levels 1 and 2, indicating precisely in the users chapters 3, 4 and 5 what operations are on which level. By doing so we obey our design objective IV.

TORRIX-BASIS LEVEL1 is a very large subset of TORRIX-BASIS (compare the headings in chapter 5; only 5.0.8, 5.9 and 5.15 are on level2). The essence of level2 is a programming strategy rather than a specific facility. Although the total array idea does not operate in full swing until level2, we yet reap many of its fruits already on level1 - where restrictions on the sizes of concrete arrays exist for assigning operations only.

2.3.3
Generating procedures, relation to ALGOL68S

We thought it wise to charge TORRIX with the task of generating the concrete arrays. Instead of leaving it to the user to apply the proper generators himself, we supply him with procedures doing it for him in the proper way (see 3.2.1). By this we solve two (unrelated) problems at once:

1. A concrete array may have any lowerbound ≥-t, and any upperbound ≤t, where t is an implementation constant (cf. section 1.2.3). That is to say, the practical total array domain extends from -t to t. This t is the maximum allowable array-subscript.

Usually [-t:t] is a far too extensive reach for practical use. This implies that there is potential danger for unintended array generation. One should not forget that many TORRIX operators (cf. 5.1, 5.2, 5.8, 5.14, 5.16 and 5.18) may generate an implicit intermediate array.

In order to enable the user to control array-generation to a certain extent, we give him the possibility of defining that extent. By a procedure-call *setgendex(1,n)* the actual reach for array-generation will be confined to [*1:n*]. In every procedure-call or operator application inducing a concrete array-generation, it will always be verified whether the bounds of the neonate concrete array lie in the thus defined interval, or not (which is then a fatal error).

Hence, array generation in TORRIX can be a strictly restrained happening; it is all in the users hand (he may even, temporary, disallow each possible generation of an array by *genallowance(false)*, or even *setgendex(0,-1)*). See also 3.2.1 and 4.3.

2. The official sublanguage ALGOL68S (cf. {37.2}) does not allow *heap*-generators - hence, it does not have a garbage-collector. In 2.3.2 we have explained why TORRIX-BASIS LEVEL1 does not require a real heap - all it may need is a last-in-first-out extension to the working stack. However, for the inevitable generation of concrete arrays within routine-texts we had - by syntactic compulsion - to apply a *heap*-generator.

Now the question may arise as to whether TORRIX-BASIS LEVEL1 can be implemented through a true ALGOL68S-system. Let us call such an implementation "TORRIX68S".

Clearly, TORRIX68S will be a proper subset of TORRIX-BASIS LEVEL1. We may also infer from the foregoing that in TORRIX68S no *heap*-generators should occur, and that for TORRIX68S a certain LIFO-extension to the working stack may be necessary. Apart from that, no TORRIX68S-program should contain a call of a generating procedure in any formula. This is one of the official sublanguage restrictions.

Our conclusion is, that an adaptation is quite feasible - though not an entirely trivial task. In fact we preluded an activity of that kind by carefully confining the direct use of a *heap*-generator to precisely three places (to wit 6.1.1,2&3). Everywhere else we call a specific procedure (i.e. *genintarray*, *genarray1* or *genarray2*) for the purpose. Calls of these procedures may best be treated as macro-applications, but the solution is at the discretion of the TORRIX68S-implementer (who presumably will link the 'prelude' to the ALGOL68S 'standard-prelude', anyhow).

2.3.4
Refers

The treatment of references (*ref*) in the orthogonal frame forms the very basis of the ALGOL68 mode concept. It certainly was a great idea to unify the concepts of reference (pointer), address (name), changeability (variable), parameter-passing (identity-declaration), storage-allocation (generator) and identifier-declaration in one basic concept. However, the idea of proclaiming non-*ref* to be the mode of the not-variable (i.e. constant) entities was - though at first and second sight quite natural and consistent - a mishap. It ignores the irrefutable fact that even constant values have a memory address and that for various good reasons we may be wanting to know that address without having the intention to change its contents. This is closely bound up with the to copy or not to copy problem in parameter-passing.

The more orthogonal construction would have been to maintain a generalized "*ref*" at the end of a *ref*-chain: we might call it a "refer"(rer). We then distinguish the four logically possible refer-states:

refin an entity to which a value may be assigned, but from which no value can be obtained;

refex an entity from which a value may be obtained, but to which no value can be assigned;

ref an entity to which a value may be assigned, and from which a value can be obtained;

fer an entity to which no value may be assigned, and from which no value can be obtained.

This is not the place to analyze the (static and dynamic) possibilities of this set-up. Suffice it to state that the *ref* above conforms to the ALGOL68 *ref* as it stands now; that the *refin*, being essentially a dynamic refer, may be used to denote a not yet defined value (i.e. a *refin amode* is an uninitialized *amode* entity); and that the *refex*, in its static quality, replaces the present ALGOL68 non-*ref*; a *fer* finally can be used for an entity that may only be trans*fer*red. For an analysis of these ideas see {21} and {22}.

The relevance of the subject for TORRIX appears from the observation that quite natural and also very efficient modes for complex vectors and matrices would become possible:

mode covec = struct(vec re,im),
mode comat = struct(mat re,im)

Observe that the *re*- and the *im*-field (the real and the imaginary part) can - in the spirit of the total array - have different domains. These may overlap, coincide or even be disjunct, and both may be the zero(-vector or -matrix). The economy of this data-structure is obvious.

Unfortunately enough this very nice construction is irreconcilable with the ALGOL68 standard-declaration *mode compl = struct(real re,im)*. What we need is:

*mode compl = struct(*refin *real re,im)*

This would perfectly agree with the above declaration for *covec* and *comat*, provided that the operators for this *compl* would then allow the *re*- and the *im*-field to be located at quite different addresses. That is precisely what the refin might accomplish, but is incompatible with the present definition of *compl*.

2.3.5

Selectors

In 3.1.3 we summarize the ALGOL68 concept of indexing ("slicing") an array, which allows the selection not only of a single element (which is essentially what "subscriptors" do) but also of subarrays (which is what "trimmers" do). Here we assume the reader to be more or less familiar with the contents of that section. Almost everything is fine with these features: they do most of the selections we would like to be done, and the notation suggests nicely what happens.

Their main disadvantage for TORRIX is that they can not be extended, which would have been possible if they were formulated as standard-operators. Suppose we had "∘" and "//" available for the purpose; an operator notation for all of the indexing would then be:

U∘i	for	*U[i]*
i∘U	for	*U[i]*
U∘(h//k)	for	*U[h:k <u>at</u> h]*
(h//k)∘U	for	*U[h:k <u>at</u> h]*
j∘A∘i and A∘i∘j and A∘(i∘j)	for	*A[i,j]*
(i∘j)∘A and i∘j∘A and j∘A∘i	for	*A[i,j]*
A∘i	for	*A[i,]*
j∘A	for	*A[,j]*
A∘(h//k)	for	*A[h:k <u>at</u> h,]*
(h//k)∘A	for	*A[,h:k <u>at</u> h]*

Here the *U* (U) and *A* (A) denote units yielding a <u>*vec*</u> or a <u>*mat*</u> respectively. The "∘" and "//" notation has the same expressive power as the ALGOL68 notation; and we did everything we could to implement it in such a manner that it becomes a serious competor (we only had to apply other symbols). The <u>*at*</u>-feature could have been implemented accordingly, but is of less importance.

There is no point in overemphasizing notational matters, though they certainly play an important part in scientific publication and discovery. The operator notation above is in good accordance with the various notational fashions in mathematics and fysics:

linear algebra:	u_i , A_{ij} , $A_{i\circ}$, $A_{\circ j}$			
differential geometry:	u_i	(covariant vector) ,	A_j^i	
(Einstein)	u^i	(contravariant vector)		
quantum mechanics	<u|	and	<u|i>	covariant
(Dirac)	|u>	and	<i|u>	contravariant
	<j|A| ,	|A|i> ,	<j|A|i>	

Subscripting ("u_i") and superscripting ("u^i") cannot easily be done in a programming language, which is why we need something like an operator (mathematically, indexing is an operation). The "∘" would be a natural symbol but is not available in ALGOL68 for that purpose. Therefore we chose the "?" for the subscriptor, thus writing:

u?i for u∘i , *i?u* for i∘u
a?i for a∘i , *j?a* for j∘a
etc.

An important TORRIX operation is the trimmer "⫽", for which we chose the double-symbol "//". It effects projection:

u?(h//k) i.e. u∘(h⫽k)
(h//k)?u i.e. (h⫽k)∘u

Both formulae project *u* on the subspace spanned by the unitvectors h upto k. This is why *u?(h//k)* is the extension of the ALGOL68 *u*[*h*:*k* at *h*] rather than of *u*[*h*:*k*], which sets a new lowerbound at *1* (ALGOL68 is slightly lower-bound *1* preferent-by-default).

Needless to say that both "?" and "//" extend the ALGOL68 concrete array indexing to total array indexing, thus yielding also the out of bounds virtual zeros.

On level2, however, we run into difficulties when we want to allow a total-subscriptor in a 'destination' (i.e. the left-hand side of an assignment). We may then be requiring non-reserved storage. Of course we can also here generate a well fitting new concrete array replacing the (not large enough) old one. But this is dangerous: the old concrete array may still be referred to by other *vec*s (or *mat*s), and this would be the beginning of chaos. For that reason we defined a separate destination-selector "¿", for restricted use only.

For further particularities on selectors and trimmers see 3.1.6, 3.3.3 and 5.0 (3, 4, 5, 7 and 8).

2.3.6

Descriptors

The ALGOL68 multiple value (concrete array) is composed of a sequence of values (its elements) all of the same mode, controlled by a descriptor which contains all necessary information concerning the subscript bounds, the physical addresses of the elements, their distance (stride) in core and whatever may be appropriate to administer the set (cf. 3.1.3). The ALGOL68-report is less specific: it describes a descriptor as a k-tuple of bound-pairs ($k \in N^+$) and it requires a 1-1 correspondence between a k-tuple of integers (each in the domain of the corresponding bound-pair) and the set of elements (cf. 2.1.3.b&c in {36}). In other words: it is not required that the elements are stored following a prescribed order (i.e. row-wise, column-wise or whatever-wise).

Letting the implementer free to follow his own taste and technical preference in such matters is a good principle but it may interfere with another good, namely the possibility of precisely describing certain quite common and well-defined selection operations. ALGOL68 has a fine set of subscripting, trimming and shifting operations on multiple values; it totally lacks provision for diagonal selection, subscript permutation etc.

It is highly improbable that any complete implementation, true to the report (specifically supporting all slicing features), would not be able to implement the other reasonable wishes. Anyhow we need them for TORRIX. The basic feature necessary for our application is an *at*-like operation which permutes the subscripts. Something like:

A[perm *1,2*]	is the same as *A* as it stands,
A[perm *2,1*]	interchanges the first and the second subscript,
A[perm *1,1*]	selects the main diagonal, following the row-indexing,
A[perm *2,2*]	selects the main diagonal, following the column-indexing.

and correspondingly for arrays with more subscripts.

We helped ourselves by declaring through pragmats (cf. 2.1.4, remark on *), what we needed for TORRIX. In terms of the absent perm-feature they could have been declared roughly as follows:

op *trnsp* = (*mat* a)*mat*: a[perm 2,1];
op *diag* = (*int* k, *mat* a)*vec*: a[,*at* 2 *lwb* a -k][perm 1,1];
op *diag* = (*mat* a)*vec*: 0 *diag* a

See 6.7 for the pragmats through which we did it. It is our conviction that we did not actually leave the orthogonal path - quite on the contrary.

For further discussion see {21} and {22}.

2.3.7

Collateral loop-clauses

Pointwise operations on arrays play the key role in the definition of all operations with *scal*, *vec* and *mat*. By "pointwise" we mean operations that are performed on the individual elements of the arrays involved, and for which the order of action is immaterial. All additive operations with *vec*s and *mat*s and all sumproducts are pointwise in the above sense.

The ALGOL68 loop-clause now prescribes precisely the order in which the operations must be done. It gives us no means of expressing that the order of action is immaterial, and precisely that may be a piece of information of crucial importance for all kinds of (compiled or hand-coded) optimizations in algebraic operations. Moreover, the ALGOL68 loop-clause does not return a value. From practically all TORRIX-operations we expect some *vec* or *mat* to be returned, and that is then in most cases precisely the *vec* or *mat* on which we operate (i.e. the logical candidate for a return value).

In a sense ALGOL68 is not orthogonal in this matter. It has a collateral-clause returning a compound value as the complement of a serial-clause; it lacks a collateral-loop-clause completing the serial-loop-clause. A syntactic form might be:

with **subscript list** *thru* **REFETY ROWS unit** *do* **serial clause** *od*

The yield of the whole construct should be the yield of the **REFETY ROWS thru part**. Observe that the collaterality applies to the multiple elaborations of the entire **do part** and not to what has to be done between *do* and *od*.

As an example, compare the declaration of a Hilbert matrix as it has to be now in TORRIX68:

```
proc hilbert = (int n)mat:
      (mat dave=gensquare(n);
       for i to n
        do for j to n
             do dave[i,j]:=1/(i+j) od
        od; dave
      );
```

with the considerably more appropriate construction:

```
proc hilbert = (int n)mat:
      with i,j thru mat dave=gensquare(n); dave
        do dave[i,j]:=1/(i+j) od;
```

The availability of precisely this kind of collateral-loop-clause would have greatly influenced the appearance of the entire TORRIX system, which would have been both more transparant and more open for various optimization techniques. We cannot change the former within the frame given by ALGOL68 - the optimization can and should be done anyway, and it may have a dramatic effect compared to the (otherwise quite reasonable) objectcode compiled by the present tools.

For an example compare the routine-text 6.18.1, for the linear transform of a vector, with the following collateral loop-clause version. Observe the various optimizations possible; because the elaboration order of the do-part is now at the discretion of the implementer:

```
op × = (mat a, vec u)vec:
         if mat aa = (lwb u//upb u)?a; zero aa
        then zerovec
        else vec v = genarray1(lwb aa, upb aa);
            with i,j thru aa
              do v[i]+:=a[i,j]×u[j] od; v
         fi
```

3. USERS GUIDE

3.1	TORRIX68	57
3.1.1	The TORRIX-ALGOL68 subset	57
3.1.2	The use of *int*, *scal* and *coscal*	59
3.1.3	Multiple values, descriptors and slices	61
3.1.4	The three TORRIX-levels	64
3.1.5	Ascription, assignation and generation	70
3.1.6	Selectors	78
3.2	LEVEL1	82
3.2.1	Generation bounds, level0-objects	82
3.2.2	The declaration of level1-objects	84
3.2.3	Interrogations	87
3.2.4	Level1 ascription and assignation	90
3.2.5	New values, new descriptors, new torrixes	91
3.2.6	Sigmas and extrema	98
3.2.7	Level1 assigning operations	100
3.2.8	Array generating additions	102
3.2.9	Sumproducts	105
3.2.10	Array generating scal-vec-mat multiplications	107
3.3	LEVEL2	113
3.3.1	The declaration of level2-objects	114
3.3.2	Level2 assignation	116
3.3.3	Destination-selectors	122
3.3.4	Trimming operations	124
3.3.5	Level2 assigning additions	126

3. USERS GUIDE

The purpose of this tutorial is to illustrate the use of TORRIX as a programming tool. More detailed and systematic information on the features discussed can be found in chapters 4 and 5 - cross references are placed in the headings of the subsections. If, after consulting the text referred to, a doubt persists, you should go to the actual source-texts which occur in the corresponding sections in chapter 6. These texts, in the last resort, decide all remaining matters of doubt.

Apart from the contents of 3.1 and the given sequence of 3.2 preceding 3.3, the order of reading is largely immaterial. The general subdivision follows the technical order of the subjects, rather than some strong didactic principle. Only 3.1 has been set up as a survey of relevant ALGOL68 features.

The user of this guide is supposed to have some knowledge of ALGOL68, though he does not need to be an expert - not even an experienced ALGOL68-programmer (he might become one by using TORRIX). For an introduction to ALGOL68 we refer to {07}, {13}, {14}, {23}, {25} and {32}.

Unless otherwise stated, all identifiers and other applied-indicators occurring outside the direct context of their declaration, will identify those in always the last version of the declarations D1, D2, D3 ... in the present chapter 3, or those in the chapters 4 and 6 (as discussed in 5), or else they have the meaning as indicated in 4.1 (notational conventions). We shall occasionally deviate from this rule in pictures. It will then always be apparent from some such picture that and how we deviated.

3.1 TORRIX68

Apart from a few operators (they are always marked with *), the entire TORRIX set of declarations can be formulated in ALGOL68 proper without contradicting any of its standard-declarations; i.e. we can make TORRIX a 'library-prelude' in the strict sense of the Report (cf. 10.1 in {36}).

We shall denote the ALGOL68 implementation of TORRIX by "TORRIX68", but without being pedantic in this matter (we shall often simply speak of "TORRIX" where "TORRIX68" might be more at its place). On the rare occasions that we want to distinguish TORRIX68 from the implementation without the *-operations, we may use the denotation "TORRIX*".

Having TORRIX68 at his disposal - preferably of course as a precompiled (and where possible optimized) library - a programmer will normally not use more than a rather limited subset of the full language, simply because TORRIX68 will provide most of what he may need for his purpose. Therefore, "TORRIX68" (or "TORRIX" for short) and in the slightly more restricted sense "TORRIX68*", in fact stands for an ALGOL68-"dialect": a large 'library-prelude' extension together with a certain subset of ALGOL68. There is no point in precisely delimiting this subset which may also depend on the particular application area.

In this section we outline those features of ALGOL68 which are indispensable. Sections 3.1.3 to 3.1.6 discuss the ALGOL68 features (and the specific TORRIX68-form of some of them) which are of particular interest to all who want to try the system. One should certainly read these sections before going to the subject matter proper in 3.2 and 3.3.

3.1.1 The TORRIX-ALGOL68 subset

4.3.1/4.3.5

The control-constructs with

if , *then* , *else* , *elif* and *fi* ,
case , *in* , *out* , *ouse* and *esac* ,
for , *from* , *by* , *to* , *while* , *do* and *od*

and their nestings, together with the concept of a serial-clause as a construct yielding a value (possibly an instance of *empty* of mode *void*), enable users to formulate the vast majority of their programs without *goto*s.

Consequently, we leave labels and jumps for what they are in this language: practically negligible. Even in the TORRIX routine-texts they never occur.

Where TORRIX is a system designed to regulate the use of multiple values for the operations of vectors and matrices in the broadest sense, it is not surprising that the programmer need not explicitly write the generators for them. TORRIX68 provides specific procedures for that purpose.

Instead of declarations such as:

1' *[1:n]real u* , *[1:m,1:n]real a*

one should always write:

1 *vec u = genvec(n)* , *mat a = genmat(m,n)* etc.

Certain protections built in the system make 1 safer than 1' (see 5.1), but more important is the greater generality of 1 as compared to 1' (see 3.2.1 and 3.2.2). The declarations 1 express, moreover, the TORRIX facts of life: that *vec* and *mat* (and also *covec*, *comat* and a few others) are standard modes and that *genvec* and *genmat* generate them in all dimensions required.

There is, of course, absolutely nothing against applying multiple value generators for objects other than *vec*s and *mat*s.

For example:

2 *[0:k]bool flag* or *[1:4]char code* etc.

For the particular application of multiples such as representing vectors and matrices and related objects, the user is strongly advised to adhere consistently to the TORRIX style of doing things. The same applies to never writing the declarers "*real*" and "*compl*" directly - see next section.

We also never write a so-called row-display for vector- or matrix-values; we do not need them - we have all kinds of operators for setting vectors or matrices (of even arbitrary sizes) to specific values (see 3.2.4 and 3.2.5).

As to the many independent ("orthogonal") other features of the language - such as the manipulation of routines (in particular via procedure- and also operation-declarations), structured modes, united modes, parallel-clauses, *bits*, *bytes* etc. - if you feel you need them, then by all means use them as far as the implementation allows.

However: be aware of the way things are done (or not done) in the TORRIX-system. This applies, in particular, to the total absence of the *flex*ible feature in TORRIX68, although this is a context where one certainly would expect it. A better way of treating multiple values of unequal, varying and even vanishing size is via a higher level of reference (i.e. modes beginning with *ref ref*). This is precisely one of the innovations of TORRIX (see also 2.3).

To the standard-prelude of the language we add a few operators the authors seem to have forgotten - they may be of much wider use than just for TORRIX and they are obvious candidates for machine-coded optimization:

4.3.1 the operators *min* and *max*
defined for *int*s and returning an *int*:
3 min 7 = 3 , *3 max 7 = 7*

4.3.5 the exchange-operator *=:=*
defined for *ref int*s returning a *ref int*,
and *ref scal*s returning a *ref scal*,
and *ref coscal*s returning a *ref coscal*:

let *i* and *j* be *ref int*s such that *i=3* and *j=7*,
then we have *i=7* and *j=3* after the operation *i=:=j*,
which returns its left operand (a *ref int*).

for *scal* and *coscal* see next section.

3.1.2
The use of *int*, *scal* and *coscal*

4.3.3/4.3.4

The plain modes *bool* and *char* and their derivatives *bits*, *string* and *bytes* do not play any special role. The same applies in a sense to the *int* values which are thus used for counting (in loop-clauses), indexing (in trimmers and subscripts) and choosing (in case-clauses). The *int* values are moreover assumed to be available as special elements belonging to the "wider" mode *scal*, i.e. *int* $\subset$ *scal* (see also the operator *widen* in 4.3.3).

The declarers "*real*" and "*compl*", however, should never show up in any TORRIX-program, not even in one of the numerous "daily life" applications of real vector-spaces. The underlying field of TORRIX is always given by the more neutral declarers "*scal*" ("scalar") and "*coscal*" ("complex scalar").

One then selects the actual scalar-field by the choice of the corresponding TORRIX version. If any, a TORRIX wherein

1 *mode scal = real* , *mode coscal = compl*

will naturally be available. One of the reasons why we have *scal* is that one and the same TORRIX programtext can also be compiled under other library-versions in which, for example:

2 *mode scal = long real* , *mode coscal = long compl*

or

3 *mode scal = short real* , *mode coscal = short compl*

or whatever sizes may be available.

Therefore, even if you are exclusively interested in real computations: never use "*real*" - use always "*scal*" instead (and "*coscal*" for "*compl*"). Why deprive your programtext from the possibility of being easily compiled for various precisions?

There is, however, much more in this general *scal*-approach. You might, for instance, wish to confine the underlying field to that of the rational numbers:

4 *mode scal = rational*

and a "rational" version of TORRIX might be available. As a matter of fact: a "rational TORRIX" is nothing more than a precompiled version of TORRIX68 under an operation-library for a mode *rational* (see 2.2.3).

You might also consider finite fields, for example:

5 *mode scal = primod*

where *primod* is some prime field. Various other fields may likewise be of interest.

Further possibilities may arise from certain restrictions of TORRIX. In a subset in which we leave out all divisions, we open the possibility of considering it as defining a module over certain rings. For example:

6 *mode scal = integral*

where *integral* is a mode preferably containing "in principle all integral values" - otherwise a *long*-enough size of the mode *int* might do (see 2.2.3).

The moral of this exposition is that TORRIX defines, in a very general way, vector spaces or modules over arbitrary fields or rings. Each particular choice of a precompiled TORRIX-version implies the choice of a specific field or ring for *scal*. That is the very reason why we express the entire underlying arithmetic in a neutral mode *scal* - we want to leave the door open as long and as far as possible.

Summing up, the standard TORRIX68 modes are:

int with *index* (see 3.2.2), *scal* with *vec* and *mat*, *coscal* with *covec* and *comat*.	the declarers "*real*" and "*compl*" are banned from TORRIX.

There is nothing particular with the modes:

bool and *bits*
char and *string* and *bytes*

which are free for any application.

3.1.3
Multiple values, descriptors and slices

Vectors and matrices - though represented by their abstractions *index*, *vec*, *mat* etc. - are eventually realized as objects of the modes []*int* (or *intarray*), []*scal* (or *array1*), [,]*scal* (or *array2*), []*coscal* (or *coarray1*), [,]*coscal* (or *coarray2*) and a few other more baroque constructs for special purposes (see volume II). The row-of-modes (or *array*s for short) are therefore important for those who want to use the library.

An *array* (or "multiple value") consists of a linearly ordered set of elements, controlled by a descriptor which contains all necessary information concerning the subscript bounds, the physical addresses of the elements, their distance (or "stride") in core and whatever else may be appropriate to administer the set.

In TORRIX we never work with the *array*s proper, we always use entities referring to them (see 3.2) or even references to such entities (the subject matter of 3.3). In the sequel, let U refer - directly or indirectly - to an

<u>*array1*</u>, <u>*coarray1*</u> or <u>*intarray*</u> and let *A* likewise refer to an <u>*array2*</u> or <u>*coarray2*</u>.

U and *A* can be sliced: *U*[*i*], *U*[*h*:*k*], *A*[*i*,], *A*[,*j*], *A*[*i*,*j*], *A*[*i*,*h*:*k*], *A*[*h*: , :*k*] etc. A <u>slicer</u> consists of an <u>indexer</u> between square brackets. It can best be conceived as an operator acting on *U* and *A* and always returning a reference to an individual element or to a newly created descriptor of an on occasion empty sub-<u>*array*</u> of its argument. Compare 3.1.6 for a more general (but usually less efficient) operator for the selection of elements and other slices. It is essential that a slicer never makes a copy of, or even touches, an element of the <u>*array*</u> it acts upon. We now summarize the facts which ought to be known to all TORRIX-users:

- an <u>indexer</u> may consist of <u>subscripts</u> (*i* in [*i*,*h*:*k*]) and <u>trimmers</u> (*h*:*k* in [*i*,*h*:*k*], but also *h*: in [*i*,*h*:], :*k* in [*i*, :*k*], : in [*i*, :] and even <u>empty</u> in [*i*,]);
- an <u>indexer</u> with <u>subscripts</u> only (such as [*i*] and [*i*,*j*]), selects one single element and does not create a new descriptor;
- an <u>indexer</u> with one or more <u>trimmers</u> creates a new descriptor for the sub-<u>*array*</u> selected, it does not copy any element;
- each <u>subscript</u> in an <u>indexer</u> decreases so to say the number of indices in the resulting new descriptor by 1 until the descriptor vanishes;
- the default values for <u>absent</u> <u>bounds</u> in a <u>trimmer</u> are the corresponding bounds in the mother-descriptor;
- all <u>lowerbounds</u> in a new dewcriptor are set to 1, except where the <u>trimmer</u> was <u>empty</u>, or where a <u>new-lowerbound</u> (<u>*at*</u> *h* in [*i*,*h*:*k* <u>*at*</u> *h*]) required a specific other value.

Since they occupy a rather central role in TORRIX68, we give a few examples of slicing operations:

<u>1</u> *U*[*i*]
returns a <u>*ref*</u> to the *i*th element of *U*;

<u>2</u> *A*[*i*,*j*]
returns a <u>*ref*</u> to the *j*th element in the *i*th row of *A*
i.e. a <u>*ref*</u> to the *i*,*j*th element of *A*;

3 $U[h:k]$
returns a ref to (a newly created descriptor of) the sub-array indicated by the trimmer $h:k$; lwb $U[h:k] = 1$, upb $U[h:k] = k-h+1$;

4 $A[i,\ ]$
returns a ref to the ith row of A,
lwb $A[i,\ ] = 2$ lwb A , upb $A[i,\ ] = 2$ upb A ;

5 $A[\ ,j]$
returns a ref to the jth column of A,
lwb $A[\ ,j] = 1$ lwb A , upb $A[\ ,j] = 1$ upb A ;

6 $A[i,h:\ ]$
returns a ref to the slice $[h:\ ]$ of the ith row of A,
lwb $A[i,h:\ ] = 1$, upb $A[i,h:\ ] = 2$ upb $A - h + 1$;

7 $A[i,h:k]$
returns a ref to the slice $[h:k]$ of the ith row of A,
lwb $A[i,h:k] = 1$, upb $A[i,h:k] = k-h+1$;

8 $A[i,h:k$ at $h]$
returns the same as 7, but now:
lwb $A[i,h:k$ at $h] = h$, upb $A[i,h:k$ at $h] = k$;

9 $A[i,$at $0]$
returns a ref to the ith row of A, but:
lwb $A[i,$at $0] = 0$, upb $A[i,$at $0] = 2$ upb $A - 2$ lwb A .

Further facts of importance for a sound understanding of TORRIX, are:

- for all subscripts i in an indexer it is required that:
  $Lwb \le i \le Upb$, where Lwb (Upb) indicates the corresponding lowerbound (upperbound) in the mother-descriptor;
- for all lowerbounds h, in a trimmer $h:k$, it is required that:
  $Lwb \le h$;
- for all upperbounds k, in a trimmer $h:k$, it is required that:
  $k \le Upb$;
- if, in any trimmer $h:k$, we have $h>k$, then the descriptor created by this trimmer is a so-called "flat descriptor", indicating an empty array (i.e. an array without elements[1]).

[1] The Report speaks of a "ghost-element" for sake of precise definition.

A particular application of a flat descriptor is:

10 *U*[*maxdex*:*mindex*], in which $mindex \leqslant 0 \leqslant maxdex$ (see 3.2.1) returns a *ref* to a flat descriptor, the set of elements of *U*[*maxdex*:*mindex*] is empty.

Empty *array*s of the kind 10 play the very special role of "zero-vector" ("zero-matrix") in TORRIX (see 3.2.2).

Observe that *U*[*h*:*h*] refers to an *array1* with precisely one element, but *U*[*h*] refers to that very element, i.e. the mode of *U*[*h*:*h*] is *ref array1*, but the mode of *U*[*h*] is *ref scal*.

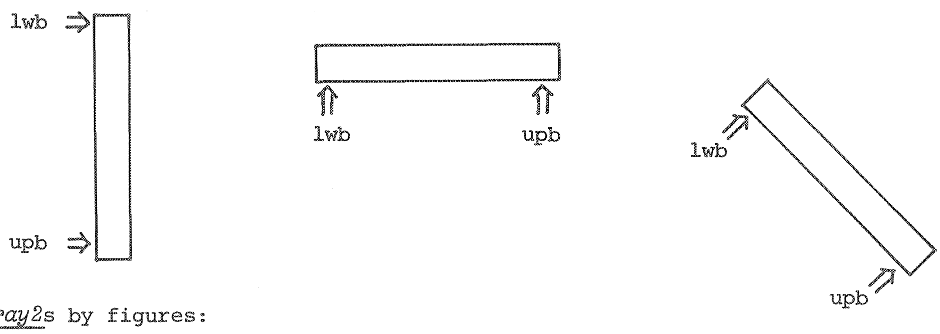
3.1.4
The three TORRIX-levels

5.0

Here and in all following sections we shall often use the term "*array1*", to imply also "*coarray1*" (and sometimes even "*intarray*") and "*array2*" to imply also "*coarray2*". By writing "*array*" we mean, as before, "*array1*" or "*array2*".

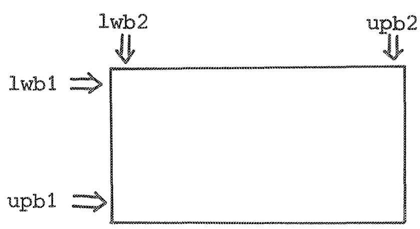
We shall depict *array1*s by figures as:

1



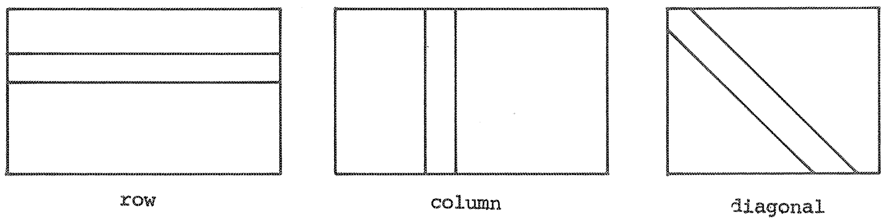
*array2*s by figures:

2



and *array1*s as a row (or a column or a diagonal) of an *array2* by:

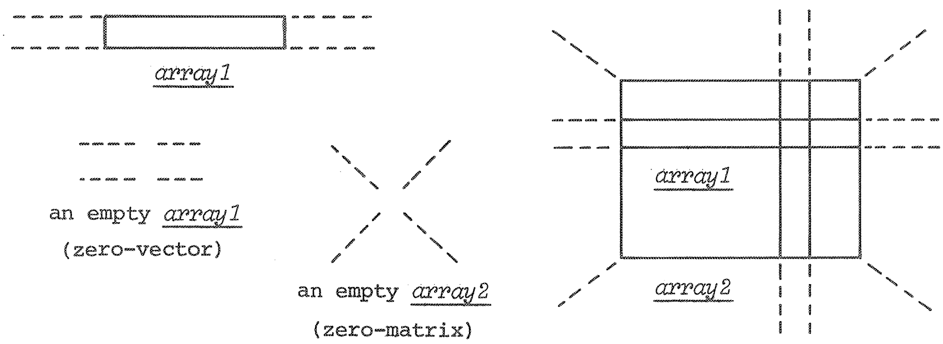
3



These "concrete" *array*s must be conceived of being embedded in much more extensive "total"-*array*s. A total-*array* consists of a concrete part - its constituent concrete *array* - together with a virtual part - consisting of a (for all purposes sufficient) number of virtual zeroes. Hence, the virtual zeroes do not exist in the memory - they are assumed by TORRIX-operators and should, therefore, always be taken into account by TORRIX-users. Virtual zeroes thus are, in a sense, very real objects although we shall never meet them in a computer memory.

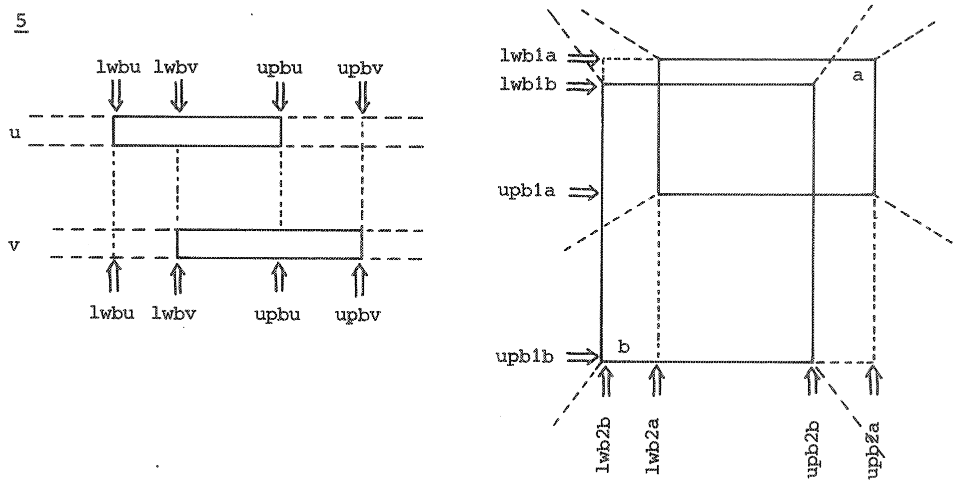
When we wish to emphasize the virtual presence of a total-*array*, we shall bring it in the picture as follows:

4



Two total-*array*s with different concrete bounds in one figure will often be depicted as follows:

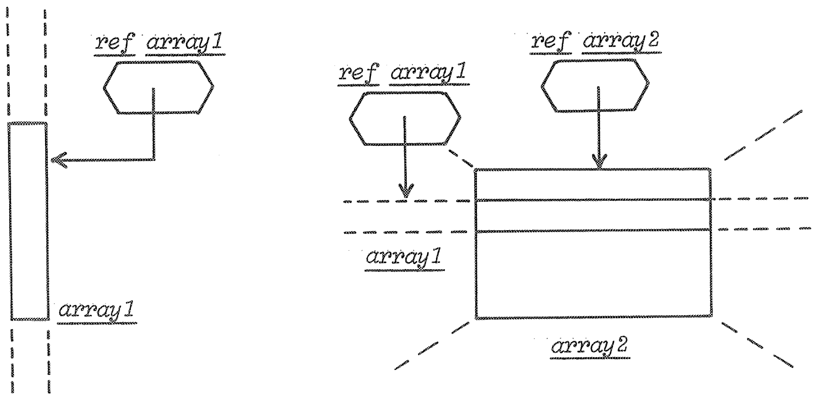
5



In TORRIX all *array1*s as all *array2*s are "compatible" ("conformable") regardless of their bounds. That is to say: total *arrays* have always the same (virtual) bounds - only the bounds of their concrete parts may differ.

We call the *arrays* level0-objects. They form the raw material of the system in which they never show up as such - they always go in (and go out) behind their references. These references (or "names" as they are called unfortunately in the Report) are the level1-objects. The relationship between level1- and level0-objects will be depicted as follows:

6



It is essential in this language that the bounds of the *arrays* do not enter the mode of their references. A *ref array1* is a *ref*[]*scal* and it can refer to concrete *array1*s of (in principle) all sizes. The same applies to the mode *ref array2* (that is *ref*[,]*scal*).

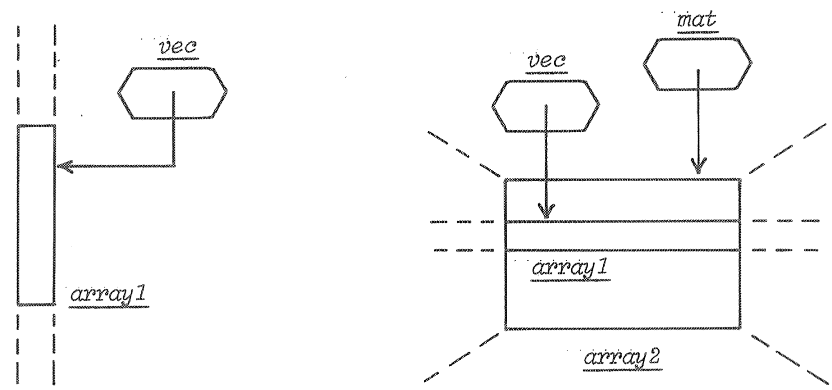
We are therefore entitled to say that *ref array*s refer to the total *array*s, even where they will be implemented as something pretty close to the core-addresses of concrete-*array* descriptors.

It is this slightly idealized concept of a *ref array* that makes a vector or matrix in TORRIX68:

mode vec = ref array1 , *mode covec = ref coarray1* ,
mode mat = ref array2 , *mode comat = ref coarray2* .

So we arrive at the following TORRIX-picture of vectors and matrices:

7



One should, from now on and in all contexts, conceive a *vec* or a *mat* as an entity consisting of:

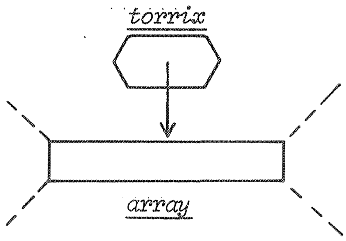
| a reference (a "name"), i.e. the *vec* or *mat* proper,

together with

| a total *array*, often called "the *array*" of the *vec* or *mat*,
| or "its *array*", or even "its value" or "its contents".

Whenever we can do so in this chapter without confusing the issues, we shall take *vecs* (*covecs*) and *mats* (*comats*) together and speak of "*torrix*es" and "their *arrays*", depicted as follows:

8



A particular kind of information about *torrix*es is given by the values of their concrete bounds. These bounds are fixed at the generation of the *array*. A *torrix* can, in principle, refer to *array*s of all sizes (expressed by saying that a *torrix* refers to a total *array*). When we actually wish to use a concrete *array* of another size, then we have to generate a new one and our *torrix* must now be able to refer to that new location.

Here we are faced with two kinds of variability:

the level1-variability of the sub-values and the individual concrete elements in the *array*,

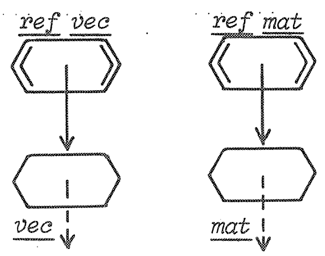
the level2-variability of the concrete bounds, i.e. the possibility of turning virtual zeroes into concrete (non-zero) elements (the concrete *array* becomes "longer") and the possibility of turning concrete zeroes into out of bounds virtual zeroes (the concrete *array* becomes "shorter").

The variability of sub-values and individual elements can be realized entirely on level1 because the scene can then be laid at the concrete *array*. The variability of the *array*-bounds must be realized on one level in reference higher (we want to shift the scene, i.e. to alter the *torrix*).

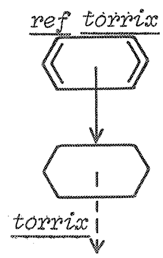
For the manipulation of level0-objects (the *array*s and their sub-values), we needed level1-objects (*ref array* such as *vec* and *mat*). In order to make them refer to "longer" or "shorter" concrete *array*s, we must be enabled to manipulate *vec*s and *mat*s (level1-objects). To that purpose we need level2-objects (*ref vec*, *ref mat* etc.).

The relationship between level2- and level1-objects will be depicted as follows:

9



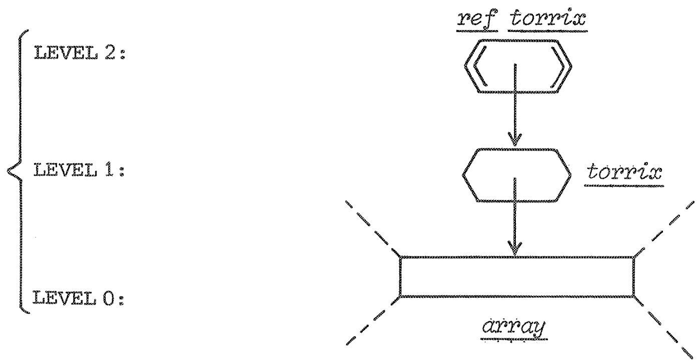
in general:
not distinguishing
*vec*s and *mat*s



Observe how in a level2-level1 relation the object referred to is a *torrix* but in a level1-level0 relation the *torrix* is the referring object.

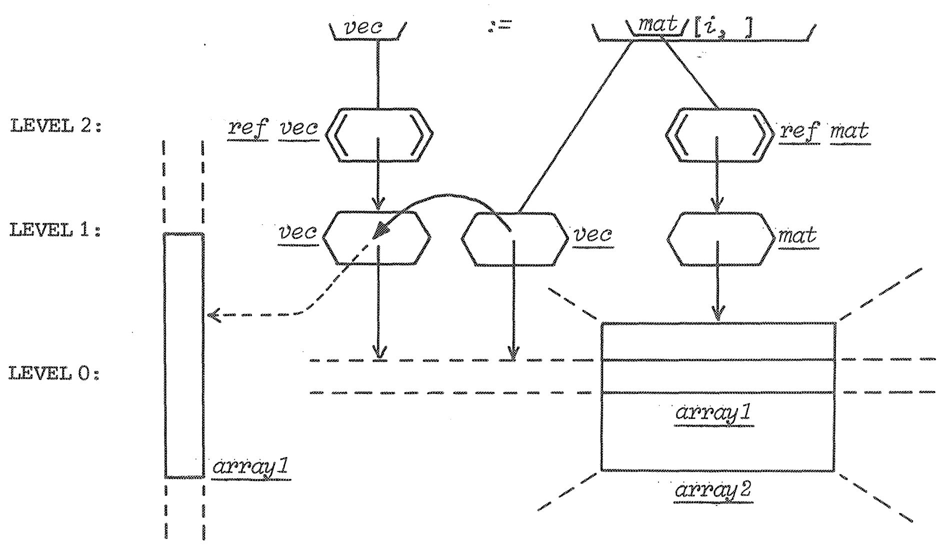
Sometimes the three TORRIX-levels show up together in one and the same compound action. This occurs when level2-assignations play a role (see 3.3.2). In our pictures we then meet schemes such as:

10



We call the reference from a *ref torrix*, via a *torrix* to an *array* a depth-reference. Depth-references lead to seemingly complicated configurations, which nevertheless reflect precisely the existing relations between objects on all the three distinct levels. As an example consider a typical TORRIX-assignation (for details see next section):

11



In this picture it is shown how, as a result of the assignation *vec* := *mat*[*i*,], the vec of *vec* ceases to refer to the leftmost array1 and is made to refer to the *i*th row of *mat*.

In the sequel we shall often make use of these semantic pictures for their self-explanatory quality.

> The reader who is not (yet) familiar with operations on different ref-levels, is advised to skip all specific level2-considerations while reading the sequel and chapter 3.2 (which is entirely on level1). Thereafter he should return to this point.

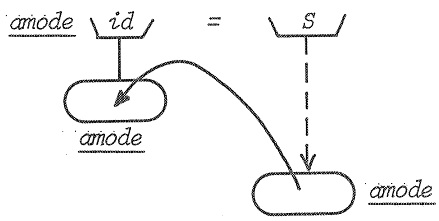
3.1.5
Ascription, assignation and generation

Ascription, assignation and generation are fundamental concepts in TORRIX as they are in ALGOL68. Let S be a unit yielding an amode value (possibly "a posteriori" after one or more coercions such as dereferencing or widening). Now S may occur in the syntactic position of a source, this means that its value will be preserved for later use. There are two ways of achieving this:

- by ascribing the value of *S* to an identifier,
- by assigning the value of *S* to a variable.

An ascription takes always place in an identifier-declaration and is quite conspicuous in an identity-declaration (a sub-class of the former):

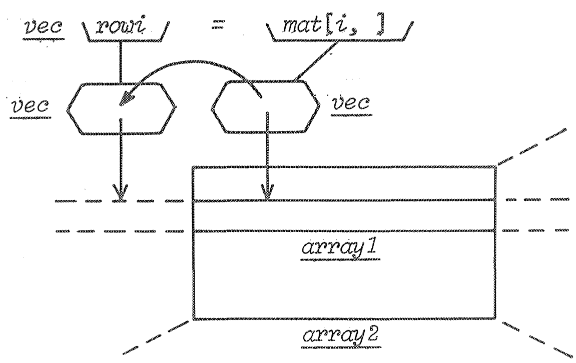
1



Here the lefthand side requires an amode value to be ascribed to the identifier *id*, i.e. to be held in a place adhered to *id* in such a way that this value becomes the "a priori" yield of *id*. The action "to be held in a certain place" will be depicted through a bowed arrow pointing into that place. This arrow may be interpreted as "making a copy of" or "transport of information". An implementer can, however, often do better.

An important example of an identity-declaration in TORRIX is:

2



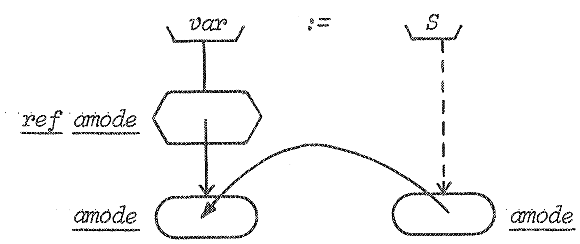
Observe that we did not depict the object yielded by *mat* itself. Regardless of its level (*mat* may be a ref mat-identifier as also a mat-identifier), when sliced it always yields a level1-object (a mat or a vec). This phenomenon is known to the ALGOL68-connoisseurs as "weak dereferencing".

The important fact of the above identity-declaration, however, is that the *array1* of *mat* [*i*,] has not been copied. Its reference ("address") has been ascribed to *rowi*, so that from now on it is also the *array1* of *rowi*. For all concrete elements of *rowi* and *mat* [*i*,] we thus have:

rowi [*j*] *is* *mat* [*i*,*j*]

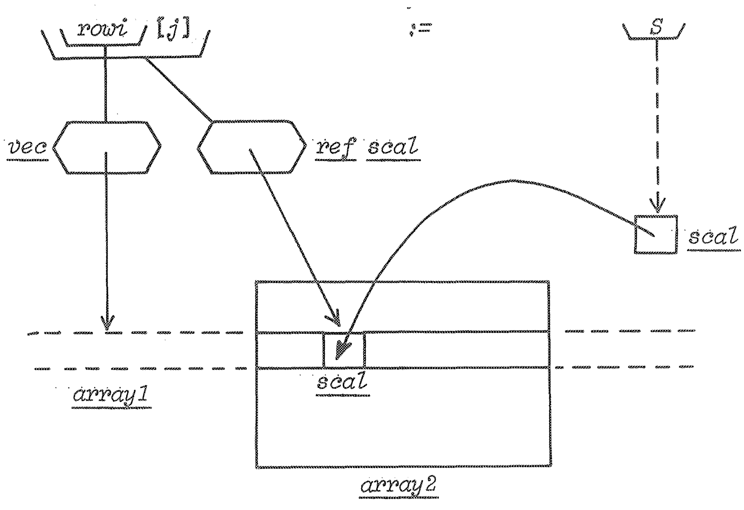
This implies, for instance, that *rowi* [*j*] and *mat* [*i*,*j*] are identical (are the same) and that any assignation to the former is that same assignation to the latter and vice versa.

An assignation always alters the contents (the "value") of a variable (unless, by chance, the new value happens to be equal to the old one). In an assignation the lefthand side - the destination - must be a *ref* to the value to be altered:

3


Observe that the a priori yield of *var* (its *ref amode*) remains the same - it is the object referred-to that becomes a copy of the righthand value.

An example of an assignation in TORRIX is:

4


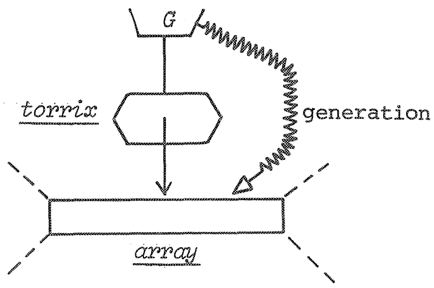
Here *rowi* is a <u>*vec*</u> (a <u>*ref*</u> <u>*array1*</u>) and therefore *rowi*[*j*] is a subscripted variable of the mode <u>*ref*</u> <u>*scal*</u>. After the identity-declaration <u>*vec*</u> *rowi* = *mat*[*i*,], the above assignation achieves the same as

$$mat[i,j] := S$$

but by simpler and more efficient means (one subscript instead of two).

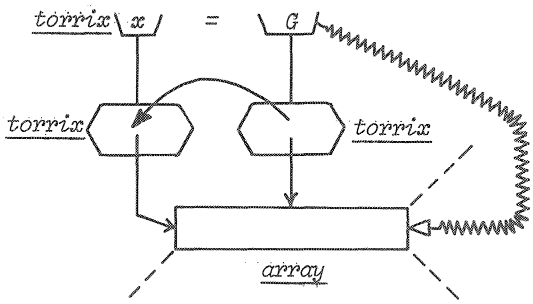
We now consider a particular kind of a source, namely a <u>generator</u> *G*. A generator reserves new space for an object of a given mode and it yields its reference (the address of the new location). As a rather general example we depict the generation of an <u>*array*</u>, yielding a <u>*torrix*</u>:

<u>5</u>



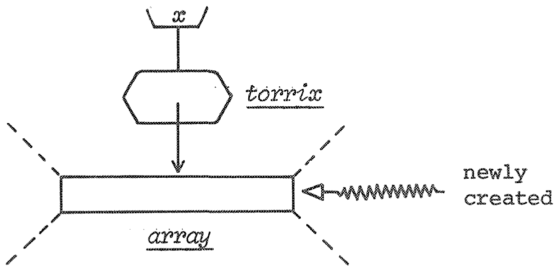
For the sake of the generality pursued in TORRIX, the generating source for an <u>*array*</u> will always be performed through a routine, the effect of which is depicted in the above figure. Hence, the generating level1-declaration of a <u>*vec*</u> or a <u>*mat*</u> will always be of the form:

<u>6</u>



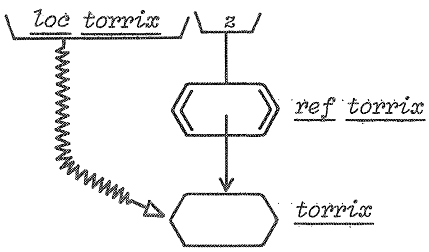
Observe how the <u>*torrix*</u> generated is being ascribed to the identifier *x*. The resulting relation is:

7



On level2 we can apply the common ALGOL68-generators because there are no bounds and there are no scope-problems either. So we can declare:

8



We shall always use the optional *loc*-symbol, in order to make a clear distinction between:

9 *torrix* $x = G$ a level1 declaration (picture 6)

and

10 *loc* *torrix* z a level2 declaration (picture 8)

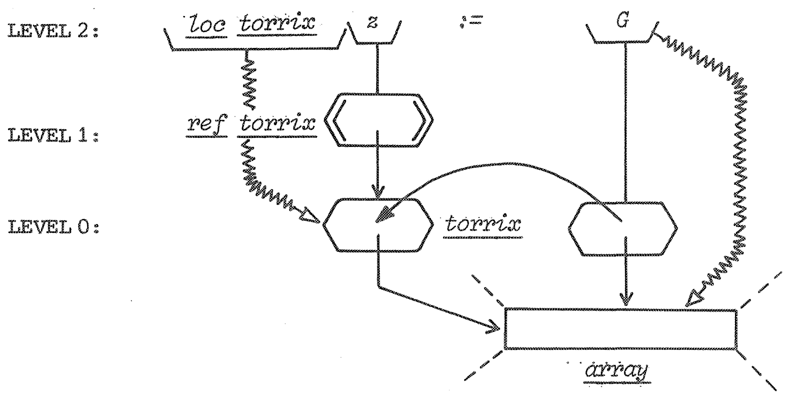
Observe that in both declarations we meet ascription and generation:

in 9 the object generated is an *array* and its *torrix* is being ascribed to x;

in 10 the object generated is a *torrix* and its *ref* *torrix* is being ascribed to z.

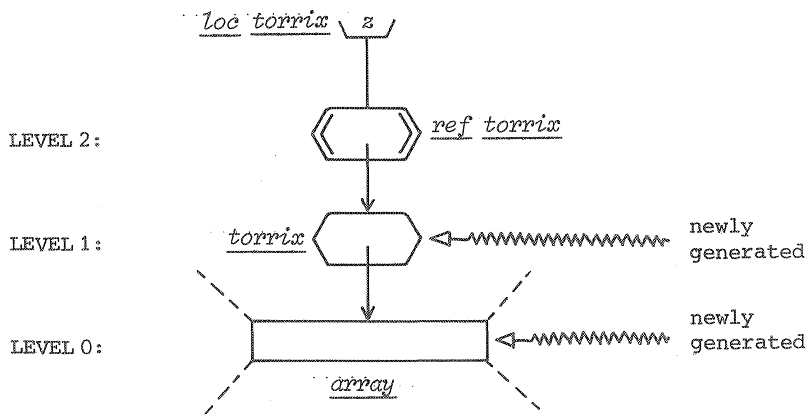
On level2 we often want to generate a location for a new *torrix*, together with an *array* to which this *torrix* has to refer initially. This can be achieved simply, in one initializing declaration:

11



It may be instructive to examine how in 11 two generations, one assignation and one ascription work together on three levels in order to establish the typical full-TORRIX relation:

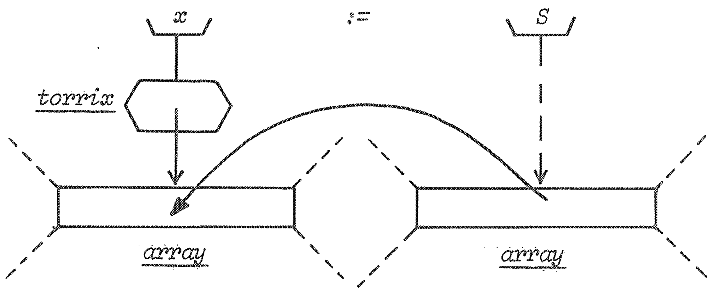
12



Of course, 12 depicts the result of the generation happening in 11.

We now have to consider the two kinds of assignation in TORRIX: namely, those on level1 and those on level2. Let x be a *torrix*-identifier (i.e. a level1 *array*-variable) and let S be any source yielding an *array*:

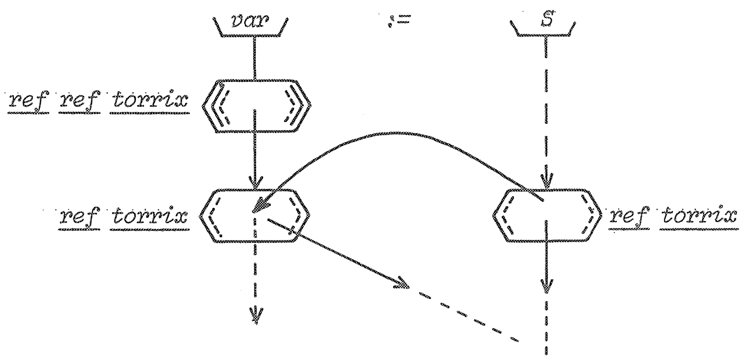
13



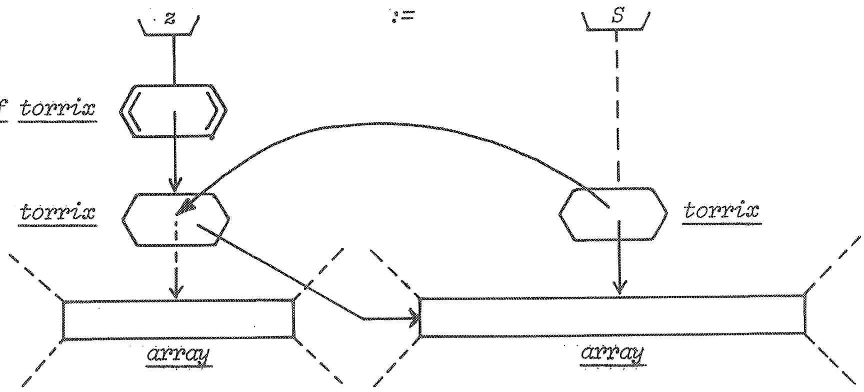
This is a typical level1-assignation. The destination, being a *torrix* (*ref array*), requires an *array*. The source, by assumption, yields an *array*. It is now required that all the bounds in the source match the corresponding bounds in the destination. Under this condition, the lefthand *array* becomes a copy of the righthand *array*.

For a level2-assignation we must remember that assignment, in ALGOL68, takes always place on the highest level possible. That is to say, even when *amode* in 3 is a *ref bmode* (and *bmode* in its turn possibly a *ref cmode*, and so on), we still have the assignment as depicted in 3:

3'

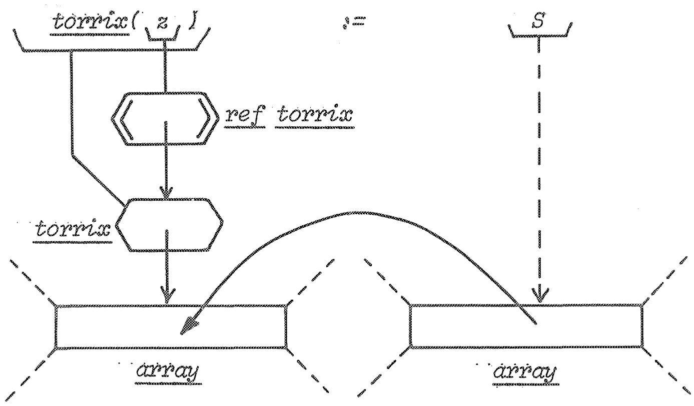


Now you should not have any difficulty in understanding the typical level2-assignation - *z* being a *ref torrix*-identifier:

14


The important point here, of course, is that the bounds of the *array*s at the left and at the right do not play any role. Even when the righthand *array* is much "longer" than the lefthand one, the action can take place as depicted. The *torrix* of *z* loses all interest in its original (lefthand) *array*, because it is now set to refer to the righthand one. And if there is no other *torrix* interested in that lefthand *array*, it may become a willing prey to the garbage-collector and disappear entirely from the memory.

It might happen that you wish to do a level1-assignation on a level2-variable. The standard way to achieve such in ALGOL68 is by means of a so-called cast:

15


In TORRIX68 we have an even simpler way of achieving the same effect and that is by slicing the level2-variable with an empty trimmer. A slicer always returns the *vec* (*covec*) or *mat* (*comat*) value of its operand (weak dereferencing). Therefore we can write:

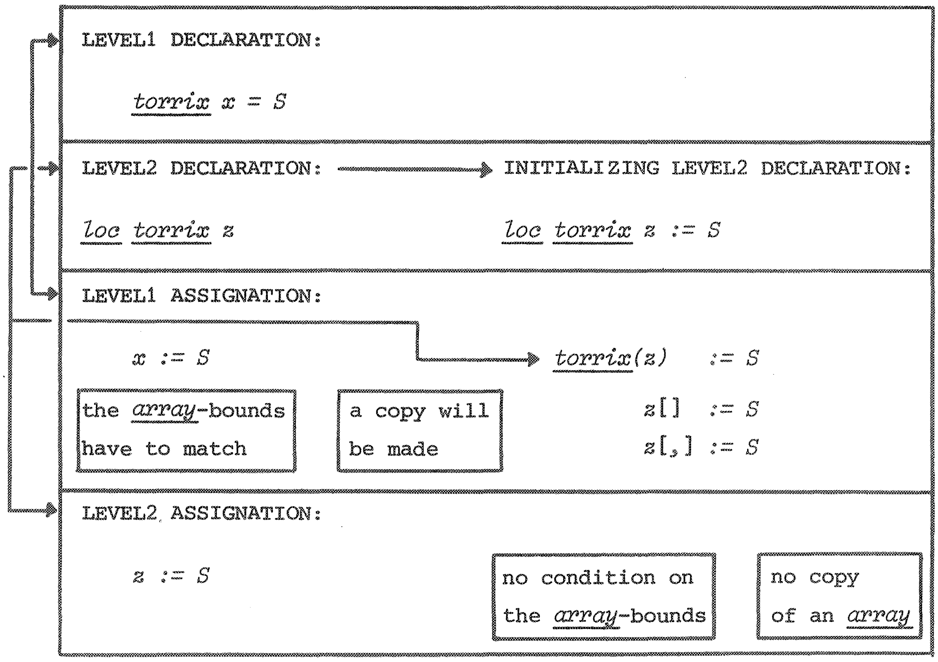
16a   *z*[] := *S*   when the *torrix* was a *vec* (*covec*)

or

16b   *z*[,] := *S*   when the *torrix* was a *mat* (*comat*)

We shall always use this technique because it follows the general style of expressing things in TORRIX. Observe that 15, 16a and 16b are level1-assignations and, therefore, all the *array*-bounds have to match.

For their central importance we finally summarize the general constructs discussed in this section. They form so to say the "main frame" of the TORRIX-system.



3.1.6

Selectors   5.0.3 to 5.0.7

Slices in ALGOL68 are syntactically built-in language features. Their expressive power became apparent in 3.1.3. Their only - but essential - limitation is, that they cannot be extended beyond the concrete *array*-bounds.

Selection of an out-of-bounds element through a slice is undefined (i.e. leads, hopefully, to a program abort with a proper error-message), and rightly so because in the general case the element simply does not exist.

In TORRIX, however, a concrete *array* is thought of as being embedded in a well-defined total-*array*, and thus it has some reality beyond its concrete bounds. Even though the virtual (out-of-bounds) elements are nowhere in the memory, their value is zero.

Therefore - and in particular for applications in volume II - one would like in certain situations to circumvent the irrevokable boundconditions of the built-in slice. The most obvious way around is to declare operators which return the concrete element(s) in case of a concrete selection, and also respond adequately to any selection beyond the concrete bounds (instead of calling out). We shall call such operators selectors.

We have to distinguish source-selectors which do no more than return zero at virtual selection, and destination-selectors which actually concretize virtual elements by extending the concrete *array* as to comprise them. Source-selectors are fairly simple. Destination-selectors, however, are rather drastic operations and we postpone their treatment until 3.3.3 where we can better understand their possible implications.

The source-selector for *u[i]* is either *i?u* or *u?i*. The essential difference between *u[i]* and *i?u* or *u?i* is that the former is undefined for i < *lwb* u and *upb* u < i, whereas the latter two return *0* for such values of i. Hence, *i?u* and *u?i* are well-defined for all i. For example:

1 *zerovec?i = 0* for all i
(*zerovec[i]* is undefined for all i).

For a trimmer we have a mode:

mode trimmer = struct(int lower, upper)

and a dyadic operator // returning the *trimmer* defined by its *int* operands.

We can now extend the slice *u[h:k at h]* to *u?(h//k)* or *(h//k)?u*, which are both defined beyond *lwb u* and *upb u*. Observe that *(h//k)* comes in the place of *[h:k at h]* (and not of *[h:k]*). The reason for this will be discussed in 3.2.5 (see also 2.3.5).

Let, for a *trimmer* example, $1<h<m$ but $m<k$ (i.e. *lwb* $u < h <$ *upb* $u < k$), we then have:

2 $u?(h//k)$ returns $u[h{:}m$ _at_ $h]$

whereas $u[h{:}k$ _at_ $h]$ would be undefined because $k>m$.

The result of selection with a _trimmer_ can also be shifted if you feel like needing such (which will not often be the case in TORRIX, see 3.2.5):

3a $(u?(h//k))[$_at_ $p]$ returns $u[h{:}m$ _at_ $p]$

In particular:

3b $(u?(h//k))[$_at_ $1]$ returns $u[h{:}m]$

Observe also:

4 $zerovec?(h//k)$ _is_ $zerovec$ for all h and k.

So far for the selectors ? and // as they have been defined for _vec_s. They truly generalize slices, i.e. without loss of expressive power.

The application of ? to a _mat_ reduces the number of subscripts by 1. Hence, $a?i$ and $j?a$ both return a _vec_:

5a $a?i$ returns $a[i,\ ]$, if 1 _lwb_ $a \le i \le 1$ _upb_ a
_is_ $zerovec$, for all other i.

5b $j?a$ returns $a[\ ,j]$, if 2 _lwb_ $a \le j \le 2$ _upb_ a
_is_ $zerovec$, for all other j.

The selection $a?i$ thus returns the _vec_ of the ith _row_ of the total-_array2_, and the selection $j?a$ the jth _column_. As a direct consequence of the definition of ? for _vec_s, we now have:

6 $(a?i)?j = i?(j?a)$ returns $a[i,j]$, if 1 _lwb_ $a \le i \le 1$ _upb_ a
and 2 _lwb_ $a \le j \le 2$ _upb_ a
returns 0 , for all other i and j.

Of course we may want to leave out (or put in) brackets: $a?i?j$, $i?j?a$, $a?(i?j)$ etc. They should all return the same _scal_. To that purpose we have a mode:

mode _pair_ = _struct_(_int_ $rowsub, colsub$)

We now define ? for two _int_ operands to return such a _pair_, and then also for a _vec_ and a _pair_ to select the _scal_ from the total-_array_. Observe that "$(i?j)$" comes in the place of "$[i,j]$". The result of this little game with ? is, that we actually achieved:

<u>7</u> $(a?i)?j = a?(i?j) = a?i?j$
$= (j?a)?i = j?(a?i) = j?a?i$
$= (i?j)?a = i?(j?a) = i?j?a = a[i,j]$ for concrete selection
$= 0$ otherwise

Finally, and quite naturally, we can define *?* as to trim also <u>mat</u>s (rows or columns) from an <u>*array2*</u>:

<u>8</u> *a?(h1//k1)* returns the total equivalent of *a*[*h1:k1* <u>at</u> *h1*,]
(h2//k2)?a returns the total equivalent of *a*[,*h2:k2* <u>at</u> *h2*]

Consequently:

<u>9</u> *(h2//k2)?a?(h1//k1)* returns the total equivalent of *a*[*h1:k1* <u>at</u> *h1*, *h2:k2* <u>at</u> *h2*]

For a further justification of notational matters, the reader should (re)turn to section 2.3. For destination-selection see 3.3.3.

3.2 LEVEL1

TORRIX-BASIS LEVEL1 consists of the operations listed as LEVEL1 in chapter 5 (i.e. all of 5, minus 5.0.8, 5.9 and 5.15). It can be regarded as the foundation of the entire system, being its smallest, still useful (and even powerful) subset. BASIS LEVEL1 is also the kernel of TORRIX because no TORRIX-combination can do without.

However, the underlying *scal*-field may be unordered, in which case all operations presupposing order lose their meaning and become undefined. The underlying *scal* system may also be restricted, for example to an ordered or unordered ring in which case all operations based on division lose their meaning. On the other hand, we do not presuppose the *scal* system to be commutative: hence, we doubled all multiplicative operations where necessary - they can all be left out for the usual *scal*-fields (rings) based on $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}_p$ and $\mathbb{Z}$.

The essentials of playing the game not higher than the level of *vec*s and *mat*s, are:

- in all assignations the *array* bounds have to match precisely (level1-assignation, see also 3.2.4),
- the only manner of holding a newly generated *array* is by ascribing its *vec* or *mat* to an identifier, i.e. via an identity-declaration (see 3.2.2),
- the only way of getting rid of an *array* is by leaving the range in which it was ascribed to its identifier, i.e. in which it occurred in an identity-declaration.

The main topic of this chapter, however, is the method of realizing the total-*array* idea which, specifically, is the subject matter of the sections 3.2.5 to 3.2.10.

3.2.1
Generation bounds, level0-objects

4.3.2/5.0.1

The ultimate bounds of all *array*s - their virtual bounds - are given by the system constants *mindex* and *maxdex*. Their absolute values are equal and as large as possible:

$$mindex \leqslant 0 \leqslant maxdex$$

No concrete lowerbound can ever be smaller than *mindex* and no concrete upperbound greater than *maxdex*.

The concrete *array* bounds have also to obey the often more restrictive condition:

mindex<=mingendex<=Lwb

Upb<=maxgendex<=maxdex

We call *mingendex* and *maxgendex* the generation bounds because a concrete *array* can not be generated beyond them. They are (hidden) system-variables which can be set and reset by calling the procedure *setgendex*.

For example, when you know beforehand that the concrete *array*s of all your *vec*s and *mat*s will stay within the domain *1* to *n*, then - to ensure optimal safety - you should call:

1 *setgendex(1,n)*

Now any (unintentional) attempt to generate an *array* outside the thus defined frame, leads to a program abort. You should be aware that *array*s may also be generated implicitly in applying the *array* generating operators +, −, × etc. (see next section).

For linear algebraic applications the lower generation bound will normally be 1, in which case the upper generation bound can be interpreted as the dimension of the vector space under consideration.

It may be that, during a certain phase of the elaboration of your program, no *array* at all should be generated - neither explicitly, nor implicitly. Then you can disallow *array* generation (for *vec*s or *mat*s) by:

2a *genallowance(false)*

A call:

2b *genallowance(true)*

resets the generation-bounds to their default-values *mindex* and *maxdex*. A new call of *setgendex* is another way of defining a new non-empty frame for *array* generation.

The mode-declarations:

mode intarray = [*mindex*:*maxdex*]*int*;
mode array1 = [*mindex*:*maxdex*]*scal*;
mode array2 = [*mindex*:*maxdex*,
mindex:*maxdex*]*scal*

define the modes of the total-*arrays*: []*int*, []*scal* and [,]*scal*. Any attempt to apply them as actual declarers - e.g. *loc array1 monstrosity* - would lead to an abortion, caused by the absolute impossible size of this *monstrosity*.

3.2.2

The declaration of level1-objects

5.0.2/5.1/5.2

The only way on level1 to get hold of a level1-object (*vec*, *mat* or *index*) is by ascribing it to an identifier, i.e. via an identity-declaration. General pattern:

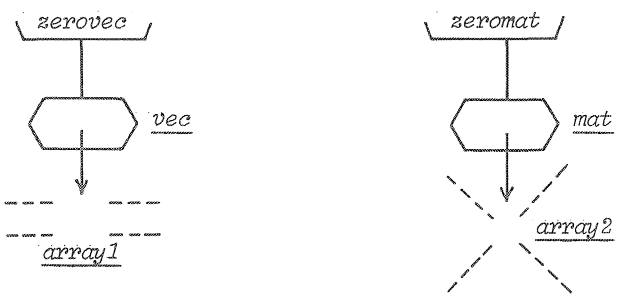
1 *torrix x* = *S*

in which the source *S* will often be a generator. Two particular and important standard identity-declarations, however, are:

vec zerovec = *heap*[*maxdex*:*mindex*]*scal*;
mat zeromat = *heap*[*maxdex*:*mindex*,
maxdex:*mindex*]*scal*

Observe that the pair *maxdex*:*mindex* defines an ultra-flat descriptor: hence, both sources generate an empty *array*. The empty vector is ascribed to *zerovec*, the empty matrix to *zeromat*:

2



The values of *zerovec* and *zeromat* consist of virtual zeroes only (compare 3.1.3.10).

The generation of non-empty *array*s takes place:

- explicitly through the procedures with identifiers beginning with "*gen*" (see 5.1),
- explicitly through the operators *copy* (or beginning with "*copy*"), *span*, *meet*, *inspan*, *subscr* and a few more special ones,
- implicitly through the operators +, −, × (depending on the mode of the operands), /, ×× and a few others (see 5.16 and 5.18).

We shall often assume the following sample-declarations:

(D1) *vec u* = *genvec(m)*,   *vec v* = *genvec(n)*,
     vec vec = *genarray1(h,k)*;

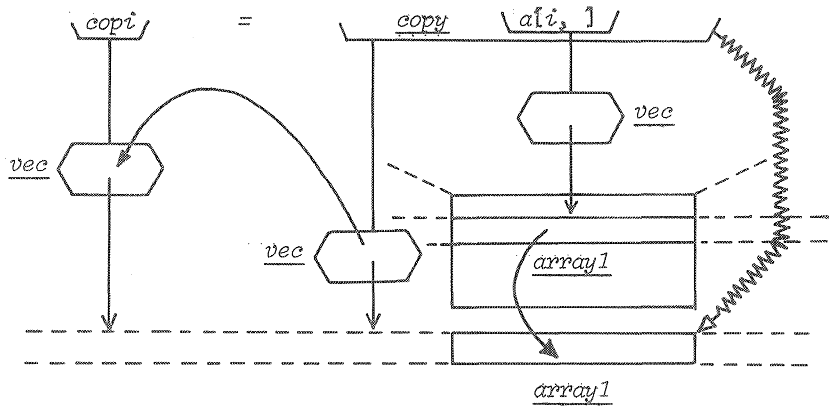
(D2) *mat a* = *genmat(m,n)*,   *mat b* = *genmat(n,k)*,
     mat mat = *genarray2(h1,k1,h2,k2)*,
     mat squ = *gensquare(n)*

All the *gen*-procedures return the *index*, *vec* or *mat* of the *array* generated. You must be aware of the non-initializing character of these *gen*-procedures. They reserve space but do not define a value (see, however, the better version of D1 and D2 in section 3.2.5). For *index* see D3 below.

Ultimately, all generations take place through these *gen*-procedures. They refuse to generate any *array* beyond the bounds defined by *mingendex* and *maxgendex*.

The operators *span* and *meet* initialize their *array*s to a zerovector or a zeromatrix. The *array*s generated through the operators *copy*, *inspan*, *subscr*, +, −, × etc. are all well-defined by their operands. Example:

3a



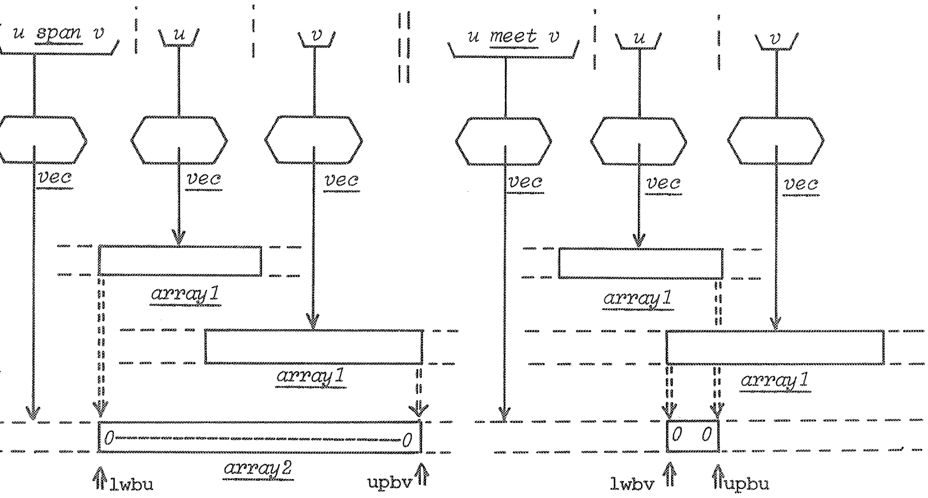
The *array1* of *copy* $a[i,]$ is a copy (new location, same value) of the *array1* of $a[i,]$. You should ascertain how

3b *vec* $rowi = a[i,]$

differs from 3a and also why both constructions are of great practical interest: 3a because we may wish to alter the (elements of the) *array1* of *copi* without even touching the *array1* of $a[i,]$, 3b because we can now access the elements of the ith row of a with one subscript instead of two. Compare also the pictures of 3a and 3.1.5.2.

The purpose of *span* and *meet* is to generate *arrays* which can contain both their operands or can be contained in them respectively.

4



The operator *span* plays its central role in the *array*-generating additions. The same applies to u *inspan* v which sets the value of u *span* v to (the value of) u, supplemented with concrete zeroes where necessary. We shall return to them in later sections (3.2.8).

The function of *subscr* is to generate and initialize an appropriate *index* for a given *vec*:

(D3) *index* p = *subscr* u

This identity-declaration generates an *intarray* with the same concrete bounds as u and its elements are set to $p[i]=i$ for all i within these bounds.

The dyadic *subscr* does the same for matrixrows (*1 subscr a*) or matrix-columns (*2 subscr a*). The function of an *index*, of course, is to keep track of permutations of elements in a *vec*, or of rows and/or columns in a *mat* (cf. 3.2.3 and 3.2.5).

Observe that this specific function of an *index* implies that the total-*array* concept has no useful interpretation for them. We shall, therefore, never speak of "virtual zeroes of an *index*" - only its concrete elements have a proper meaning.

3.2.3 Interrogations

5.3/5.4

According to the two kinds of variability (see 3.1.4), we distinguish bound-interrogations and value-interrogations.

Among the bound-interrogations the operators *lwb* and *upb* are nothing new, they are ALGOL68 standard operators. We listed them in 5.3 for sake of completeness only. Observe that *lwb zerovec = maxdex* and *upb zerovec = mindex*.

Apart from such anomalous cases, the only useable interpretation of the values returned by *lwb* and *upb* is that they are the concrete bounds of the *array* in question. You may sometimes need them for technical reasons (setting up a loop-clause, for instance).

The operator *size* returns the number of concrete elements in an *array1*. When used dyadic, its left operand specifies whether the row-size or the column-size is required from its right operand; *size* can normally be interpreted as dimension.

The operator *square* finally finds out whether its operand is a square matrix. Observe that, from a TORRIX point of view, *1 size mat = 2 size mat* is not enough for *mat* to be square. The matrix must also be centered around the main diagonal, hence *1 lwb mat* and *2 lwb mat* must be equal. You will not often need this operator, if ever. Nevertheless its judgement is worth to be known - we want to be very orthodox in these matters.

Pay some attention to the following examples:

1 $\underline{lwb}(u\ \underline{span}\ v) = (\underline{lwb}\ u\ \underline{min}\ \underline{lwb}\ v)$
$\underline{upb}(u\ \underline{span}\ v) = (\underline{upb}\ u\ \underline{max}\ \underline{upb}\ v)$
$\underline{lwb}(u\ \underline{meet}\ v) = (\underline{lwb}\ u\ \underline{max}\ \underline{lwb}\ v)$
$\underline{upb}(u\ \underline{meet}\ v) = (\underline{upb}\ u\ \underline{min}\ \underline{upb}\ v)$
(correspondingly for *mat*s)

2 $\underline{size}\ u = m$, $\underline{size}\ v = n$, $\underline{size}\ vec = k-h+1$
$1\ \underline{size}\ a = m$, $2\ \underline{size}\ a = n$

3 $\underline{size}\ zerovec = 0$ (!)
$1\ \underline{size}\ zeromat = 0$, $2\ \underline{size}\ zeromat = 0$ (!)

More important for the practice of TORRIX-programming is the operator *fitsin* returning *true* when its left operand (*vec* or *mat*) can be added into its right operand without generating a new *array*. More precisely: *x* *fitsin* *y* returns *true* iff all the bounds of *x* *span* *y* coincide with the bounds of *y*. For on the left an *index* and on the right a *vec* we have:

4 $u\ \underline{fitsin}\ v$ iff $m \leq n$
$vec\ \underline{fitsin}\ v$ iff $1 \leq h\ \underline{and}\ k \leq n$

The value-interrogations go into the values of the elements of the *array*, and some of them stand for rather non-trivial questions.

Of particular importance is the operator *zero* which decides whether its operand (a *vec* or a *mat*) is *zerovec* or *zeromat*. These two specific zero-*array*s have an ultra-flat descriptor (see 3.2.2) in order to achieve that for every *vec* (irrespective of its bounds) *zerovec* *span* *vec* and *vec* have the same concrete bounds. Correspondingly for *zeromat* *span* *mat* and *mat*.

The importance of this requirement follows readily from the observation that *zerovec* is the only vector belonging to all vectorspaces. The span of *zerovec* and an arbitrary other *vec* must therefore correspond to precisely the space of this *vec* and to nothing more.

TORRIX achieved this goal by setting the concrete bounds of *zerovec* so as to make:

5 $\underline{lwb}\ zerovec\ \underline{min}\ \underline{lwb}\ vec = \underline{lwb}\ vec$
$\underline{upb}\ zerovec\ \underline{max}\ \underline{upb}\ vec = \underline{upb}\ vec$ $\Big\{$ for all concrete *array1*s referred to by *vec*.

It will be clear that the only concrete bounds for *zerovec* satisfying this condition are *maxdex* for the lowerbound and *mindex* for the upperbound. The convention for *zeromat* follows this measure.

As a direct consequence, the value returned by any *array*-generating routine will be *zerovec* or *zeromat* whenever one of the required upperbounds is less than its corresponding lowerbound (i.e. whenever a flat descriptor shows up, cf. 5.1).

The operator *zero* now provides the right answer in the matter of zeroness of *vec*s and *mat*s. In the TORRIX-belief there is only one of each and these two have to be every inch a zerothing. This is a much stronger requirement than just having zero-sizes or zero-elements.

A related operator raising questions about TORRIX-orthodoxy is "=". When are two vectors equal? In the sense of numerical analysis (i.e. the underlying field is supposed to be R), one will usually define one or more suitable norms and try the possible equality of vectors and matrices according to this norm. The choice of the norm(s) depends on the application area and does not fall under TORRIX.

There is, however, a more pure mathematical standpoint which may be of interest, in particular when the underlying field is not R (but say Q or Z_p). Two vectors are equal iff all their elements are equal. That is - translated into TORRIX - the *array*-elements must be equal where the *array*s meet and all other (concrete or virtual) elements have to be 0. For a more fundamental discussion of this matter, see chapter 1 on the total-*array* equivalence class of *array*s. Precisely this is what $u=v$ finds out about u and v.

Observe that u=*zerovec* returns *true* when all concrete elements of u (if any) are 0, and that *zero* u implies u=*zerovec* but not the other way around. There are many *vec*s arithmetically equivalent to *zerovec*, but there is only one proper zerovector.

The operator *equ* in p *equ* q (p and q being *index*es) requires the bounds of p and q to match and all the elements of p to be equal to the corresponding elements of q.

More interesting are p *compat* u, which finds out whether all the elements of p can serve as a subscript for u, and k *search* p which returns the smallest subscript of p where we find the value k. Both *compat* and *search* may be of help in algorithms in which vector-elements, matrix-rows and matrix-columns are interchanged and *index*es keep track of the permutations.

Observe that, immediately after D3 *index* p = *subscr* u, we have p *compat* u - a condition that should hold during the entire computation. But for *lwb* $u \le k \le$ *upb* u we have only initially k *search* $p = k$. That situation will alter at each permutation of which the *index* had to keep track (see also 3.2.5).

3.2.4
Level1 ascription and assignation

The fundamentals of these actions have been treated in 3.1. We recall the main facts:

- in ascribing a *vec* or a *mat* to an identifier, we do not copy the source-*array* (unless the source does specify a copy) - we make the identifier yield a reference to that *array*;
- in assigning the value of a *vec* or a *mat* to a (vector- or matrix-)variable, the source-*array* will be copied into the *array* of the destination provided that the bounds match precisely.

Typical level1-ascriptions in the context of D1 and D2 are:

1	*vec copi* = *copy a*[*i*,]	the source specifies a copy (*copy* generates a new *array*).
2	*vec rowi* = *a*[*i*,]	the source does not specify a copy (no new *array*).
3	*mat copa* = *copy a*	the source specifies a copy.
4	*vec w* = *u span v*	the source does not specify any copy, but generates a new *array* the *vec* of which is ascribed to *w*.
5	*vec diaga* = *diag a*	*diaga* refers to the main diagonal of *a* (see next section), (no copy).

We now have the following typical level1-assignations, which we consider in the context of the above declarations:

6	*copi* := *v*	does not alter anything in *a* .
7	*rowi* := *v*	assigns *v* to the *i*th row of *a* .
8	*a*[*i*,] := *v*	achieves in all respects the same as 7 *rowi* := *v* .
9	*u* := *a*[,*j*]	assigns a copy of the *j*th column of *a* to *u* .

<u>10</u> u $:= v$ correct only when $m=n$.

<u>11</u> a $:= b$ correct only when $m=n=k$.

<u>12</u> $w[:h] := u[:h]$ assuming $h \leq m$.

Amusing is the situation with *zerovec* and *zeromat*. Because their modes are <u>*ref*</u> <u>*array*</u>, they are both allowable at the left of an assignation. So, at first sight, you might think that you could change them. However, the ALGOL68 rule that in a multiple-value-assignation all the bounds have to match precisely, takes away this threat: the only <u>*array*</u>s assignable to *zerovec* (*zeromat*) are the values of *zerovec* (*zeromat*) themselves:

zerovec := *zerovec*　　　　*zeromat* := *zeromat*

These are effectively the only assignations possible to a *zero*-thing.

3.2.5
<u>New values, new descriptors, new torrixes</u>　　　　5.5/5.6/5.7/5.8

There are essentially two ways of obtaining a new <u>*torrix*</u> from an existing one:

- by <u>assigning a new value</u> to (a subset of) its <u>*array*</u>,
- by <u>specifying a new descriptor</u> for (a subset of) its <u>*array*</u>.

For the former we have <u>assignation</u> (3.2.4), for the latter <u>slices</u> and <u>selectors</u>(see 3.1.3 and 3.1.6). For both we also have specific operators covering cases where assignation and selection fail or are less appropriate.

The operator <u>*into*</u> provides for a variety of assignments of <u>*int*</u>s and <u>*scal*</u>s to <u>*vec*</u>s and <u>*mat*</u>s, thus enabling the user to initialize newly generated (or to reset already existing) <u>*array*</u>s. In dislike of non-initialized locations we should improve on D1 and D2 and (in principle) always write declarations such as:

D1 <u>*vec*</u> *u* = 0 <u>*into*</u> *genvec(m)* , <u>*vec*</u> *v* = 0 <u>*into*</u> *genvec(n)* ,
<u>*vec*</u> *vec* = 1 <u>*into*</u> *genarray1(h,k)* ;

D2 <u>*mat*</u> *a* = 0 <u>*into*</u> *genmat(m,n)* , <u>*mat*</u> *b* = 0 <u>*into*</u> *genmat(n,k)* ,
<u>*mat*</u> *mat* = 1 <u>*into*</u> *genarray2(h1,k1,h2,k2)* ;

Observe that the widening from *int* to *scal* becomes "automatic" (i.e. is taken care of) through the operator *into*.

Useful possibilities may arise from the *into* version with on its left a *proc* returning *scal*. The following examples may speak for themselves.

Let be declared:

```
1     proc count = (int k)int: k ;

2     proc fac   = (int k)scal:
                 case k+1
                   in 1,1,2,6,24,120,720,5040,40320,
                      362880,3628800,39916800,479001600
                  out   if k<0
                      then torrix(fatal, "fac(negative)");
                           skip
                      else k×fac(k-1)
                        fi
                 esac # automatic widening assumed #;

3     proc facdenom = (int k)  scal: 1/fac(k) ;

4     proc hilbert  = (int h,k)scal: 1/widen(h+k-1) ;
```

In the context of the above declarations we now consider:

```
5     vec expowser = facdenom into genarray1(0,n)
```

The vector *expowser* has thus been defined to represent a polynomial of degree *n*; its elements coincide with the first n+1 coefficients of the powerseries for the exponential function.

```
6     mat testmat = hilbert into gensquare(n)
```

The square matrix *testmat* is initialized to the values *testmat*[*i*,*j*] = 1/(i+j-1).

```
D3    index p = subscr u
```

The operator *subscr* generates and initializes a companion-*index* to the vector *u*. For its use see further in this section. An *index* can be reset to its original value by:

```
7     count into p
```

Observe that in all these assignments the operator *into* finds by itself the bounds of the *array* it operates upon.

The exchange-operator $=:=$ permutes the *array*s of its operands; the bounds have to match. Usually you may wish to keep track of the entire permutation history. To that purpose we have *index*es:

D4 *index old = 1 subscr mat*

A complete row exchange-operation now consists of two actions:

8	*mat[i,] =:= mat[j,];*	exchanging two rows of *mat*;
	old[i] =:= old[j]	exchanging the corresponding *int*s of *old* (see 3.1.1 and 4.3.5).

After an arbitrary number of such exchange-operations, the new arrangement of the rows is given by *mat[i,]* , *i* running from *h1* to *k1* (compare D2). The original arrangement of these rows is preserved in *old*, so that *mat[old[i],]* reflects the original order. Observe how *subscr* in D4 sets the bounds of *old* according to *mat*.

The operators *into* and $=:=$ affect the *array*s of their operands. We may, however, obtain new *torrix*es from existing ones without even touching the elements. This is exactly what a slicer does (compare 3.1.3) and, in a more general sense, the selector // (compare 3.1.6).

For an example of this important principle we assume, for the concrete case, $1<h\leq k<n$ and we consider the concrete slice *v[h:k at h]* and the total slice *v?(h//k)*. From the TORRIX point of view it is wrong to conceive *v[h:k at h]* or *v?(h//k)* to be "a part of *v*". It is another vector:

9 *v[h:k at h]*
v?(h//k)
$\Big\{$ define the projection of *v* on the space spanned by the unit vectors *h* up to *k*.

For a clear apprehension of this essential TORRIX relation you should carefully verify the following (compare 3.1.6):

- the concrete elements of *v?(1//h-1)* and *v?(k+1//n)* may or may not be zero;
- the virtual elements of *v?(h//k)*, in particular also the elements *v?(h//k)?(1//h-1)* and *v?(h//k)?(k+1//n)*, are all zero.

Now consider the following declaration:

D5 *vec unitvec = 1 into genvec(1)*

Apparently, the *array1* of *unitvec* consists of precisely one concrete element with subscript 1 and value 1, namely *unitvec*[*1*]. All its other elements are virtual, hence zeroes.

By declaring one such unitvector we have declared in fact all possible unitvectors. The slicer [*at i*] produces them at demand for all *mindex* ≤ *i* ≤ *maxdex*. Hence, the *k*th unitvector will be returned by:

10 *unitvec*[*at k*]

We can now also say that the vector *v*[*h:k at h*] or *v?(h//k)* is the projection of *v* on the vectorspace of *unitvec*[*at h*] *span unitvec*[*at k*].

The reason why we did not declare D5 in TORRIX68 is that this one, quite contrary to *zerovec*, is not absolutely safe. Its (only) concrete element can always be superseded by a value ≠ 1. There is, unfortunately, no defense against assignations like *unitvec*[*1*]*:=s* which could obviously contaminate the very quality of being a unitvector. Of course you are perfectly free to declare and use D5 in your own programs, taking then your own responsibility.

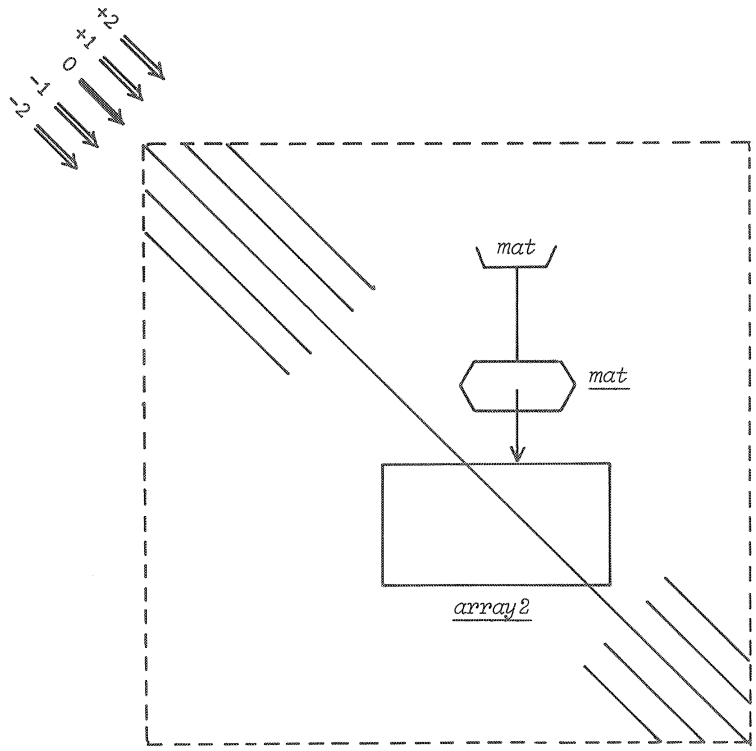
Important and interesting are the operators *trnsp*, *diag*, *col* and *row*. Being specific selectors, they are all of the non-generating new-descriptor-only type. They return a reference (i.e. a *vec* or *mat*) to that new descriptor.

A stain on them, however, is that they can not be expressed in ALGOL68 proper, which is why they have been marked with an *. Fortunately, it is an easy job to provide these routines on every ALGOL68 system that rightly implemented all the "official" slicing operations (without which TORRIX would substantially loose its flavour, anyhow).

With these operators, in particular, you must be well aware of the position of the main diagonal in total-*array2*s. A total-*array2*, extending from *mindex* to *maxdex* in its row- and column-index, is always square. The main diagonal of a total-*array2* is, by the most natural definition, the total-*array1* consisting of those virtual- and concrete elements of the *array2* which have equal subscripts: [*i*,*i*].

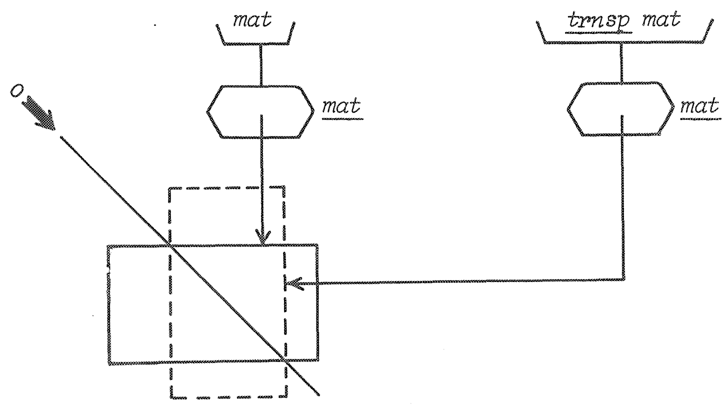
Observe that the concrete *array2* of an arbitrary *mat* may very well be in an excentric position with respect to its main diagonal. Observe also how we numbered the adjacent diagonals - counting positive for "right above" and negative for "left below":

11



Now you should have no difficulty in understanding what *trnsp* does. It returns the *mat* referring to a new descriptor which describes the *array2* elements of its operand in such a way that the row- and column-indices are interchanged. Seemingly, *trnsp* turns the *array2* over its main diagonal. In fact, of course, the elements remain where they are. They get a second descriptor which looks at them in the main diagonal mirror:

12

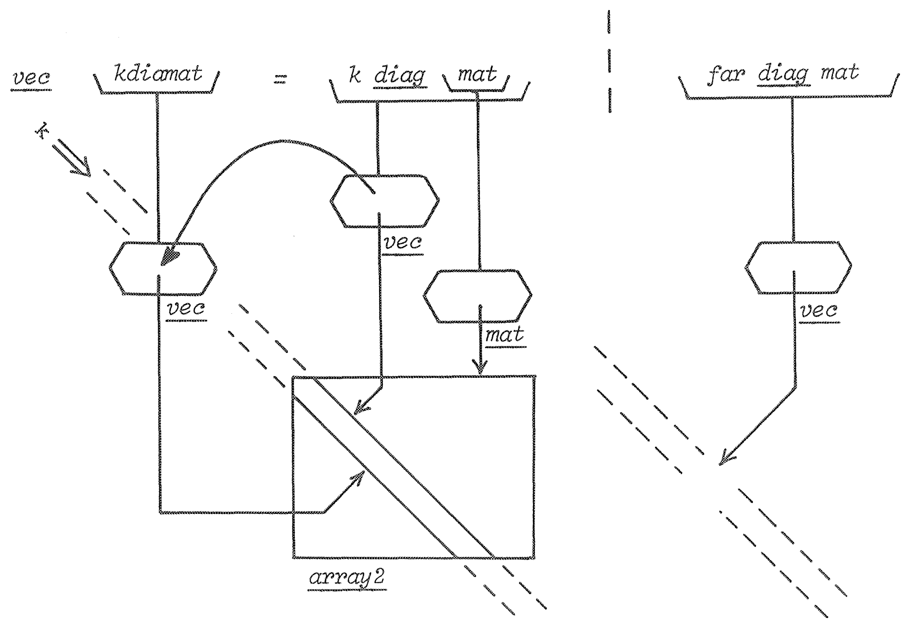


13 *mat trnspmat* = *trnsp mat*

After this declaration we have *trnspmat*[*i*,*j*] *is* *mat*[*j*,*i*], i.e. *trnspmat*[*i*,*j*] and *mat*[*j*,*i*] are one and the same element. The *array2* has got two descriptors: one referred to by *mat* and a second one referred to by *trnspmat*.

The *diag* operator returns the diagonal vector required - the left operand decides which one and the right operand yields the *array2* from which the diagonal has to be taken:

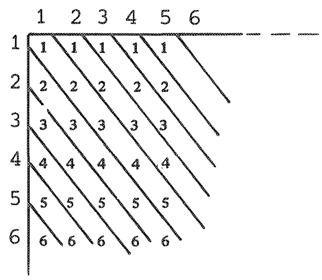
14



Observe how *far diag mat* returns *zerovec* when all *mat*-elements with subscripts [i,i+far] are virtual, i.e. when the diagonal is too far away. Consequently, all diagonals for *mindex* ≤ *far* ≤ *maxdex* exist, though most of them are *zerovecs*.

There is no natural numbering for the diagonal elements. We have chosen for the row-index of the matrix:

15



The monadic *diag* returns the main diagonal:

16 *vec main = diag mat*

achieves the same as *vec main = 0 diag mat*. Observe that *main*[*i*] *is mat*[*i*, *i*].

With the aid of *diag* we can now easily construct, for example, an identity-matrix of arbitrary size (say *n*):

17 *mat iden = 0 into gensquare(n);*
1 into diag iden

For 17 we have an operator which does it directly:

D6a *mat iden = identy gensquare(n)*

If you prefer another diagonal to be the one with ones (say number *k*), then you write:

D6b *mat idenk = k identy gensquare(n)*

In TORRIX a vector is not a matrix-with-one-row (sometimes called a row-vector), nor a matrix-with-one-column (or column-vector). A vector can be both and even more. A vector has no specific orientation - though it may get one by its position in certain formulas (see 3.2.10).

18 *vec rowi = mat*[*i*,] this *vec* represents a row of a *mat*, but it is a *vec*,

19   *vec colj = mat*[,*j*]   this *vec* represents a column of a *mat*, but it is a *vec*,

20   *vec main = diag mat*   this *vec* represents a diagonal of a *mat*, but it is a *vec*.

There are, however, situations in which you may wish to conceive a vector specifically as a column or as a row (for applications see 3.2.8 and 3.3.5). For that purpose we have the operators *row* and *col*:

D7a   *mat rowu = row u;*
mat colu = col u

These declarations achieve that the *array1* of *u* gets, in addition to its original *array1*-descriptor [*1:m*], two *array2*-descriptors: *row u* refers to a [*1:1,1:m*] descriptor and *col u* to a [*1:m,1:1*] descriptor.

As the result of D7a, both *rowu* and *colu* are *mat*s - the former a matrix-with-one-row, the latter a matrix-with-one-column. The monadic version of *row* and *col* sets the new subscript to *1*. For another subscript-value we have a dyadic version:

D7b   *mat hrowu = h row u;*
mat kcolu = k col u

which achieves the same as:

D7b'   *mat hrowu = (row u)[at h,];*
mat kcolu = (col u)[,at k]

We want to emphasize again that none of the operators *trnsp*, *diag*, *col* and *row* makes a copy of any *array*. They confine themselves to the making of a new descriptor. Of course you can make a copy with the aid of the operator *copy*, if you need one. In that case it is recommended to use the operators 5.8, because these can be expressed in ALGOL68.

3.2.6
Sigmas and extrema

5.10/5.11

Though they are closely related, there is an important and even fundamental distinction between the operations listed in 5.10 (Summation and total extrema) and those listed in 5.11 (Concrete extrema). In 5.10 the operands are truly conceived as total-**array**s and their virtual zeroes take their (virtual) part in the computations. In 5.11 we confine ourselves to the concrete **array**s. Observe that the 5.10 operators are all monadic, those in 5.11 are dyadic.

The operators **sigma** and **sigmabs** do not occur in 5.11. They return the sum of the (absolute) values of the **array**-elements. In the nature of things virtual zeroes do not add anything to these sums. Therefore, though the accumulation is (of course) confined to the concrete **array**, the result applies also for the total-**array**. This is why we have listed them under 5.10. A typical concrete application is:

1 **scal** *mean* = **sigma** *u* / **size** *u*

A typical algebraic application may be:

2 **vec** *rownorm* = *genvec(m)*;
for *i* **to** *m* **do** *rownorm*[*i*] := **sigmabs** *a*[*i*,] **od**

In the matter of finding extrema, it makes an essential difference whether we take virtual zeroes into account or not. This becomes strikingly clear in finding the minimum of the absolute values of **array**-elements. We have to examine them all if we confine ourselves to the concrete **array**: at least one of the elements has the minimal absolute value and one of them is the left-most (if there is more than one).

3 **loc** **int** *leftmost*;
scal *least* = *leftmost* **minabs** *u*

This dyadic 5.11 **minabs** operator returns the absolute value of the absolute smallest element in the concrete part of *u*. Observe that $least \geq 0$. It assigns its subscript to *leftmost*. Hence we know where to find the absminimal element.

The scene changes drastically when we want to know the **minabs** of a total-**array**. Here is absolutely nothing to examine: it can only be zero and the left-most is found at the (virtual) position *mindex*. You get a warning when you apply this monadic **minabs**: why ask for a value that can only be zero?

3.2.7
Level1 assigning operations

5.12/5.13

A level1-assigning operator-symbol consists of the token "+", "-", "×" or "/" defining the kind of operation, immediately followed by "<" or ">" pointing at the "into-operand" - the other operand will be called the "from-operand". The operations +< (plus from), -< (minus from), ×< (times from), /< (divided from), ×> (times into), /> (divided into), add, subtract, multiply or divide their from-operand into their into-operand (which is always a *vec* or a *mat* or, for + and - , may also be an *index*). They return, in all cases, their into-operand as modified by the operation - the from-operand remains unchanged.

When both the into-operand and the from-operand are *torrix*es (always of the same kind), then it depends on the kind of operation whether a certain condition must be fulfilled or not.

In the additive operations *x*+<*y* and *x*-<*y*, the from-operand must fit in the into-operand (*y* *fitsin* *x* must return *true*). The addition or subtraction is then performed elementwise.

The elementwise vector-multiplication *u*×>*v* is unconditional. The operands are the total-*array*s and the result of the multiplication from an eventually concrete non-zero into a virtual zero will clearly be a virtual zero again. The elementwise division from a vector into a vector must be treated with more care in order to prevent the undefined division by a virtual zero. In *u*/>*v* it is the vector *v* that must fit in *u*.

Observe that in *x*+<*y*, *x*-<*y* and *x*/>*y* always the right operand has to fit in the left operand. The penalty in all cases is a fatal-error program abort.

Typical examples of the use of level1-assigning operations are:

1 *loc int here;*
u /< *(here maxabs u)*

which normalizes *u* according to the maxabs-norm; *here* keeps the index in *u* where we now find the value 1.

2 *squ* /< *(here maxabs diag squ)*

which does something the like to a square matrix (see 3.2.2.D2).

3 u -< (*sigma* u / *size* u)

which subtracts from all (concrete) elements of u the arithmetic mean of u (compare 3.2.6.1). Let:

4a *vec* w = *genvec(m)*

be a vector of weighting factors for u. We can now modify u, accounting for the weighting factors:

4b w×>u;
u -< (*sigma* u / *size* u)

5a a[i,] ×< (a[j,1]/a[i,1]);
a[j,] -< a[i,]

which multiplies the ith row of a with a certain factor and then subtracts this ith row from the jth row - hence, a[j,1]=0 after the completion of 5a.

5b a[i,] ×< (a[j,1]/a[i,1]);
a[j,] +< *neg* a[i,]

which does the same to a[j,], but also turns a[i,] into its negative. Though we are not particularly fond of "one-liners", we mention that 5a and 5b can be formulated as follows:

5a' a[j,] -< (a[i,]×<(a[j,1]/a[i,1]))
5b' a[j,] +< *neg*(a[i,]×<(a[j,1]/a[i,1])), or even better.

As has been said before, the underlying *scal*-field need not be commutative. This is why we have the *scal*-into-*vec* assigning multiplication ×>, next to the *vec*-from-*scal* multiplication ×<. An example is:

6a (a[j,1]/a[i,1]) ×> a[i,];
a[j,] -< a[i,]

6a' a[j,] -< ((a[j,1]/a[i,1])×>a[i,])

The result of 6a is the same as by 5a when the *scal*-field is commutative. If not, however, then again a[j,1]=0 after the operation, but all other elements of a[j,] may have other values as compared to the result of 5a.

For improved versions of 5 and 6 compare 3.2.10.1 and 3.2.10.2 which leave a[i,] unchanged.

3.2.8

Array generating additions

5.14

In an array assigning operation, one of the operands will always be changed. The array generating operations are in this respect their contrary. From the earliest times in mathematical notation, one expects the operands to remain as they are in expressions of the form $X+Y$, $X-Y$, $X \times Y$ and X/Y. This is also why we used symbol compositions less committed to tradition, such as "+<" and "x>", for the array-assigning operations.

Now, when the operands are not to be changed, the result of the operation must be stored elsewhere. For that purpose we have to generate a new *array*. By that we get, moreover, a degree of freedom we were lacking in the array-assigning *torrix*-to-*torrix* operations where the from-operand had to fit in the to-operand. In the array-generating operations all *vec*s and *mat*s will be compatible ("conformable") in the sense that they can be combined in additions, subtractions and multiplications, irrespective of their bounds.

In this section we shall confine ourselves to the array-generating additions, i.e. to operations of the form $x+y$ and $x-y$ where x or y are both *vec*s or *mat*s. For the array-generating multiplications see section 3.2.10.

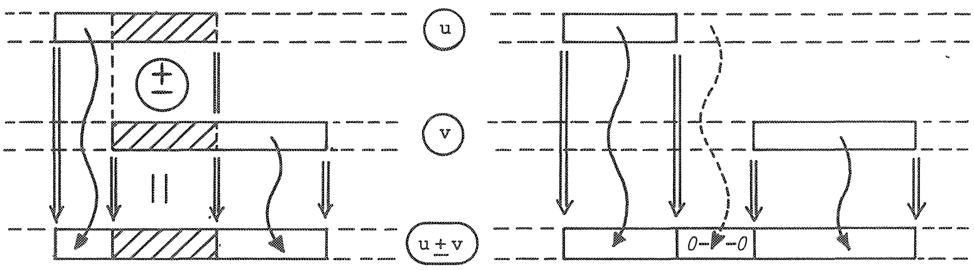
It will be clear without further discussion that - in order to make both operands fit in - we have to generate their span (see 3.2.2). Leaving aside technical details we can therefore say:

$x + y$ is semantically equivalent to $(x$ *inspan* $y)$ +< y
$x - y$ is semantically equivalent to $(x$ *inspan* $y)$ −< y

The operation x *inspan* y generates the span required and initializes it to the value of x - i.e. its element outside the concrete bounds of x are set to 0. Thereafter, y is added into or subtracted from that neonate *array* (compare 5.2 and 5.12).

For *vec*s we thus arrive at the level0-pictures:

1

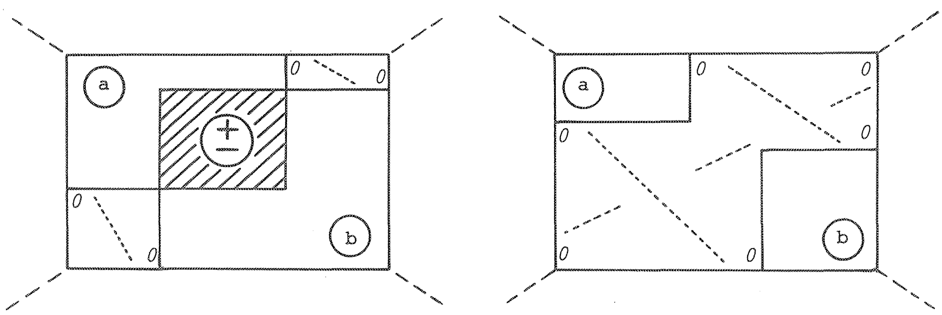


You should observe the following points:

- de facto addition will be performed on the meet of the two *array*s only (shaded in the picture),
- concrete zeroes may show up when the meet is empty (right picture),
- if one of the operands is *zerovec*, a copy (or the opposite) of the other operand will be returned as the result value,
- for *zerovec-vec* we may also write the monadic *-vec* (5.14.3),
- the operations + and - are defined for all possible *vec*s; one should, however, be aware of the size of the span generated.

Mutatis mutandis the same applies to the array-generating additions with *mat*-operands. Level0-pictures for them are:

2



Examples of usage are:

3 $\underline{vec}\ w = u+v$

in which the $\underline{array1}$ generated by $u+v$ is being ascribed to w.

4 $\underline{mat}\ c = a-b$

in which the $\underline{array2}$ generated by $a-b$ is being ascribed to c.

In using a level1-assignation you should be well aware of the size of the $\underline{array}$ resulting from $x+y$ or $x-y$:

5 $\underline{vec}\ uplusv = u\ \underline{span}\ v;$
$\underline{mat}\ aplusb = a\ \underline{span}\ b$

and now you can assign:

6 $uplusv := u+v;$
$aplusb := a+b$

On level1, however, it is better to ascribe the result of $x+y$ or $x-y$ to a new identifier than to use an assignation. In an ascription you do not have to worry about sizes and bounds. Moreover, and this is more important, in an ascription there is no copy involved. In the implementation of a level1-assignation the making of a copy is practically unavoidable. Hence, as compared to 3 and 4, 6 is inferior on two essential points: it takes more time and it requires more temporary space.

An interesting specific application of $x+y$ is its use to concatenate $\underline{vec}$s:

7 $\underline{vec}\ uv = u + v[\underline{at}\ m+1]$ (compare D1)

which ascribes a sum-vector of length m+n to uv, so that (after 7) we have $uv[1:m]=u$ and $uv[m+1:m+n]=v$.

With the aid of the operators $\underline{col}$ and $\underline{row}$ (see 3.2.5) we can also construct, from a matrix a, new augmented matrices au (extension with a column) and av (extension with a row):

8 $\underline{mat}\ au = a + (n+1)\underline{col}\ u;$ $\quad$ { a, u and v remain what they are
$\underline{mat}\ av = a + (m+1)\underline{row}\ v$ $\quad$ { au and av are new $\underline{mat}$s

Observe that we now have $au[\ ,n+1]=u$ and $av[m+1,\ ]=v$. Or, to put it differently, u has been copied into the $n+1$th column of au and v into the $m+1$th row of av.

It is even so that the operations *mat* + *h* row *vec* and *mat* + *k* col *vec* are well defined for arbitrary bounds of *mat* and *vec* and for all possible *h* and *k*. You should verify this statement by drawing a few appropriate pictures.

Where the operator col is a *operator (see 3.1, introduction), it may be that, on some implementations, one must write *a* + *k* copycol *u* instead of the more straightforward *a* + *k* col *u*. The former (with copycol) achieves externally exactly the same as the latter, but at the price of a copy operation which can be saved in case col is available. Observe that *a* + *h* row *u* is pure TORRIX68.

Although the array-generating additions are, by the modes of their operands, true level1-operations (i.e. they do not require any operand beyond level1), they will exert their full power not below level2. We shall come back on them in 3.3.

3.2.9
Sumproducts

$\boxed{5.17}$

Under this title we combine four operations with the common property that they accumulate a scal result from a sequence of products of scals. Sumproducts occur in a wide range of applications, such as:

- the definition of norms for vecs and mats (inproducts: row × column) ,
- the computation of linear transforms (matrix × column , row × matrix) ,
- the composition of linear transformations (matrix × matrix) ,
- the product of polynomials (convolution-product or Cauchy-product) ,
- the value of a polynomial in a given point (Horner-product) ,
- the composition of polynomials .

Let, here and in the sequel, the index i in $\sum$ notations run from *mindex* to *maxdex* (implying that -i runs from *maxdex* to *mindex*). In this context it becomes essential that *mindex* = -*maxdex*.

Let $\bar{\phi}_i$ be the complex conjugate of ϕ_i. Observe that $\bar{\phi}_i=\phi_i$ when the underlying field is not complex. This is tacitly assumed for in principle all TORRIX-BASIS systems. For complexification you have to use TORRIX-COMPLEX.

It will be clear that in all sumproducts $\sum\upsilon_i\phi_i$, $\sum\upsilon_i\bar{\phi}_i$, $\sum\upsilon_i\phi_{-i}$ etc. - where the multiplicands are (concrete or virtual) elements of vectors *u* and *v* - de facto multiplication takes place only with concrete elements from the meet of *u* and *v*. If their meet is empty, then all sumproducts of *u* and *v* return *0* without performing even one multiplication. Sumproducts are not only in full accordance with the total-***array*** concept, they are also efficient in their computation.

The sumproduct *u×v* returns the value of $\sum\upsilon_i\phi_i$. In TORRIX-BASIS the operator × can be used as an alternative notation for the inner product. Strictly speaking, however, the operation × is the primitive for the definition of matrix-vector, vector-matrix and matrix-matrix multiplications.

The true innerproduct *u<>v* returns the value of $\sum\upsilon_i\bar{\phi}_i$. In TORRIX-COMPLEX we thus have *u<>v = u ×* ***conj*** *v*. A nicer notation, of course, would have been "*<u,v>*" or "*u·v*", but neither of them can be defined in ALGOL68 - "*u<>v*" is a reasonable compromise.

When the underlying field is R or C, then you can define a Euclidean vector norm as follows:

1a
```
op enorm =   (vec  u)scal: sqrt(u<>u)
```

1b
```
op enorm = (covec cu)scal: sqrt(re(cu<>cu))
```

The Frobenius-norm of a matrix can be defined as:

2a
```
op fnorm =   (mat a)scal:
    sqrt((loc scal frob := widen 0;
          for i from 1 lwb a to 1 upb a
           do frob+:=a[i,]<>a[i,] od;
         frob)
        )
```

2b
```
op fnorm = (comat a)scal:
    sqrt((loc scal frob := widen 0;
          for i from 1 lwb a to 1 upb a
           do frob+:=re(a[i,]<>a[i,]) od;
         frob)
        )
```

An example of concrete-*array* application is given by 3. Let w be a vector of weighting factors and u a vector of corresponding measurements (compare 3.2.7.4a). We now have:

3 *scal mean* = (w<>u) / *sigma* w

The reverse sumproduct u><v returns the value of $\sum \upsilon_i \phi_{-i}$ - i.e. the sumproduct of all elements of u and v with opposite indices. The operation >< is the primitive for the definition of the Cauchy-product of polynomials. Observe that u[*at* k]><v = u><v[*at* k] and that both return the sumproduct $\sum \upsilon_i \phi_j$ with i+j=k. Compare 3.2.10 on the cauchy-product of polynomials.

In the Hornerproduct u *o* s, where s is a *scal* (or a *coscal* in TORRIX-COMPLEX), the left operand is now definitely conceived to represent a polynomial (implying *lwb* $u \geq 0$) or a rational function (in which case *lwb* u may be <0). Observe that *upb* u is the (highest) degree of the polynomial (rational function) and that the set of polynomials is, naturally, a subset of the "rational functions".

The formula u *o* s returns the value of the polynomial (rational function) for s ("in the point s"). Observe that:

4 u *o* 0 is defined only when u represents a polynomial (*lwb* $u \geq 0$) and then returns the value υ_0, which is 0 when *lwb* $u > 0$ and u[0] otherwise; u*o*0 = u?0.

5 u *o* 1 = *sigma*-u for all u

6 u *o* (-1) = $\sum (-1)^i \upsilon_i$ for all u

Of course we did not write these equalities to suggest that they are equally good for practical use. You should certainly write *sigma* u and not u *o* 1 and program the appropriate loop-clause instead of writing u *o* (-1).

3.2.10
Array generating scal-vec-mat multiplications

5.16/5.18

As we have seen in 3.2.8, the *vec* and *mat* operands in array generating operations are always compatible, irrespective of their bounds. The result of the operation will be stored in the newly generated *array* with bounds as required by the operands. In this section we shall consider the multiplicative operations which generate an *array*.

The multiplications with a scalar $s\times x$, $x\times s$ and x/s return a *torrix* with the same bounds as x. We thus have $s\times x$ equivalent to $s\times\!>(copy\ x)$ and $x\times s$ to $(copy\ x)\times\!<s$ and also x/s equivalent to $(copy\ x)/\!<s$. Of course we have $s\times x = x\times s$ for commutative fields.

We can now improve on 3.2.7.5a and leave $a[i,\ ]$ unchanged. Moreover we can do it in one formula:

1 $\quad a[j,\ ] -\!\!< a[i,\ ]\times(a[j,1]/a[i,1])$

The improvement of 3.2.7.6a is:

2 $\quad a[j,\ ] -\!\!< (a[j,1]/a[i,1])\times a[i,\ ]$

Compare also:

3 $\quad u\ /\!< sqrt(u<>u)$

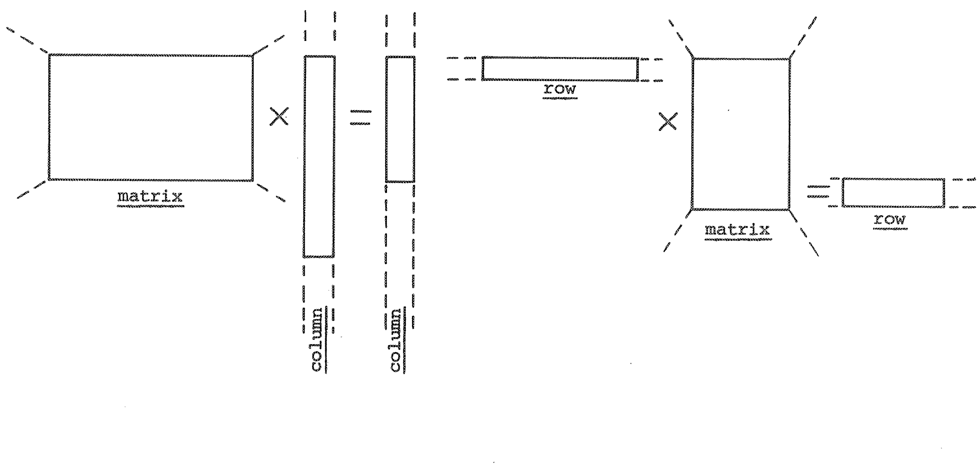
which replaces u with its normalization, and

4 $\quad$ *vec normu* $= u/sqrt(u<>u)$

which leaves u unchanged, but ascribes its normalized value to *normu*.

We now come to the matrix-vector multiplications *mat×vec* and *vec×mat*. In the former the *vec* right-operand will be conceived as a column and *mat×vec* returns a "column"; in the latter the *vec* left-operand will be conceived as a row and *vec×mat* returns a "row". These are the only cases in which *vec*s will be understood to have a particular orientation (compare 3.2.5 on the operators *col* and *row*). We thus arrive at the following level0-pictures:

5



You can be sure that $\underline{lwb}(mat\times vec) = 1\ \underline{lwb}\ mat$ and $\underline{upb}(mat\times vec) = 1\ \underline{upb}\ mat$ and, correspondingly, $\underline{lwb}(vec\times mat) = 2\ \underline{lwb}\ mat$ and $\underline{upb}(vec\times mat) = 2\ \underline{upb}\ mat$, as it should be. Now it is worth your while to assure yourself that the total-*array* concept again functions correctly and that it does so because of the way the sumproducts $mat[i,\ ]\times vec$ and $vec\times mat[\ ,j]$ come to their results.

Observe that, in the context of the declarations (D1) and (D2), the following assignments are correct:

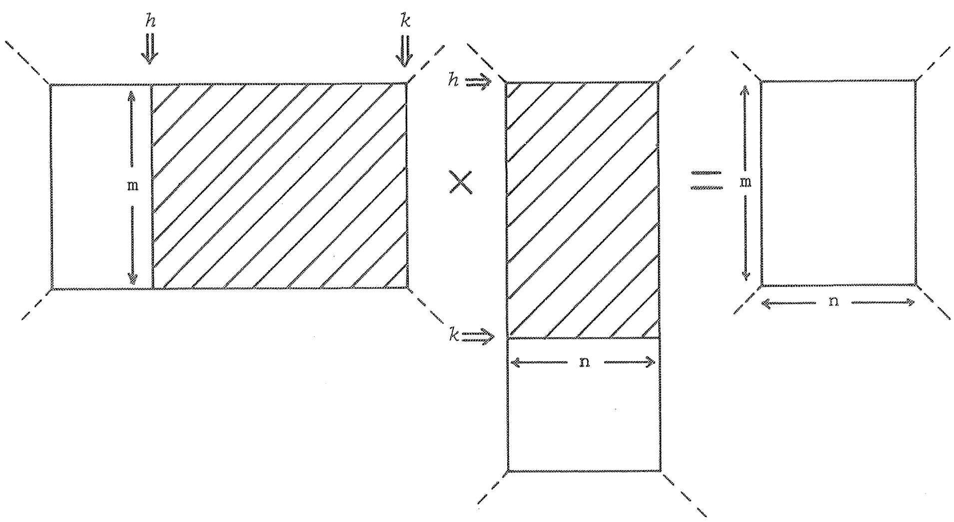
6 $\quad u := a \times v$

7 $\quad u := v \times \underline{trnsp}\ a$

"Correctness" here, of course, refers specifically to the assignation. Formulas such as $a \times v$ and $v \times \underline{trnsp}\ a$ are always correct, irrespective of the concrete bounds of their operands. The bounds of the source-result, however, must be equal to the corresponding bounds of the destination - as in all level1-assignations.

The operation $mat1\times mat2$ links up smoothly with the above operations through the same principles, *mat1* and *mat2* being *mat*s with arbitrary concrete bounds. Observe how and why the shaded parts in the picture below are the only components actually involved in multiplications - irrespective of the concrete values $2\ \underline{lwb}\ mat1$ and $1\ \underline{upb}\ mat2$.

8



Consequently, we also have:

9	*(mat×(k <u>col</u> vec))[,k]*	is equal to	*mat×vec*	for all *k*
10	*((h <u>row</u> vec)×mat)[h,]*	is equal to	*vec×mat*	for all *h*
11	*(<u>row</u> u × <u>col</u> v)[1,1]*	is equal to	*u×v*	
12	*(<u>col</u> u × <u>row</u> v)[h,k]*	is equal to	*u[h]×v[k]*	for all *h* and *k*

These equalities are of little or no practical significance (with the possible exception of 12). They demonstrate, however, the consistency of the TORRIX system.

The monadic operators *<u>trnspmul</u>* and *<u>multrnsp</u>* serve to optimize two particular matrix-multiplications in TORRIX68* (where *<u>trnsp</u>* is not available):

13	*<u>trnspmul</u> mat*	optimizes	*(<u>trnsp</u> mat) × mat*
14	*<u>multrnsp</u> mat*	optimizes	*mat × (<u>trnsp</u> mat)*

Finally we have the operations ××, *<u>o</u>* and *<u>deriv</u>* which suppose their *<u>vec</u>*-operands to represent polynomials (lwb≥0) or rational functions (lwb<0). Let be declared:

15 *<u>mode</u> <u>poly</u> = <u>vec</u>*

This mode-declaration defines "*<u>poly</u>*" to be just another word for "*<u>vec</u>*". The modes *<u>poly</u>* and *<u>vec</u>* are identical, but for polynomial applications it may be nice to name them "*<u>poly</u>*" - both are *<u>ref</u> <u>array1</u>*. For convenience sake we shall not distinguish polynomials and rational functions in their mode indication, but reserve the identifiers *p* and *q* for polynomials (*<u>lwb</u> p ≥ 0* and *<u>lwb</u> q ≥ 0*) and *r* (*r1* and *r2*) for rational functions (allowing *<u>lwb</u> r* to be *<0*).

The operation *p××q* (or *r1××r2*) returns the *<u>poly</u>* (rational function) representing the product-polynomial of *p* and *q* (*r1* and *r2*). The product *p××q* is known as the <u>Cauchy-product</u> of *p* and *q*. We thus have for all *s* of mode *<u>scal</u>*:

16 *(p××q) <u>o</u> s = (p <u>o</u> s)×(q <u>o</u> s)*

The operation *p××n* (or *r××n*) returns *p* (or *r*) to the power *n*, i.e. for all *s* of mode *<u>scal</u>* we have:

17 *(p××n) <u>o</u> s = (p <u>o</u> s)××n*

Observe that, apart from trivial cases, n must be ≥ 0. In both 16 and 17, if p or q (or both) represents a rational function, then s is supposed to be $\neq 0$ (fatal error, if not).

The functional composition $\underline{o}$ of two polynomials $p\ \underline{o}\ q$ (or $p\ \underline{o}\ r$) returns the *poly* (rational function) representing the result of the substitution of q (or r) in p, so that, for all s of mode *scal* we have:

18 $\quad (p\ \underline{o}\ q)\ \underline{o}\ s \;=\; p\ \underline{o}\ (q\ \underline{o}\ s)$

Apart from trivial cases, at least p must represent a polynomial.

Equations 16, 17 and 18 express equality on the strong assumption that the underlying field-arithmetic is exact. If, in particular, the underlying field is $\mathbb{R}$, as approximated by (some length of the mode) *real*, then there is no doubt that $(p\times\times q)\ \underline{o}\ s$, $(p\times\times n)\ \underline{o}\ s$ and $(p\ \underline{o}\ q)\ \underline{o}\ s$ will accumulate considerably more round-off errors than their practical equivalents $(p\ \underline{o}\ s)\times(q\ \underline{o}\ s)$, $(p\ \underline{o}\ s)\times\times n$ and $p\ \underline{o}\ (q\ \underline{o}\ s)$ respectively. Moreover, the latter will always crushingly defeat the former in efficiency.

Therefore, equations 16, 17 and 18 should be conceived as fixing the semantics of $\times\times$ and $\underline{o}$. Nevertheless, if the representation of the underlying *scal*-field can be exact (e.g. $\mathbb{Z}_n$ or $\mathbb{Q}$), then the operations $p\times\times q$, $p\times\times n$, and $p\ \underline{o}\ q$ may become important, and even for *mode scal = L real* there may be valuable applications.

The operation *deriv* returns the derivative of its *poly*-operand, which can algebraically be defined for polynomials as well as for rational functions. A direct, but clumsy, way of obtaining the derivative of a *poly*, say r, would be:

```
19    proc count = (int k)scal: widen k;

      poly deriv r =
      ((count into genarray1(lwb r, upb r))×>(copy r))[at lwb r -1]
```

The operator *deriv* does it more straightforward, thus showing a much better runtime-performance. The result, of course, is:

20a $\quad$ *poly* $p \;=\; (\psi_0, \psi_1, \psi_2, ---, \psi_{n-2}, \psi_{n-1}, \psi_n)$

20b $\quad$ *deriv* $p \;=\; (\psi_1, 2\times\psi_2, 3\times\psi_3, ---, (n-1)\times\psi_{n-1}, n\times\psi_n)$

For a rational function r we get accordingly:

<u>21a</u> <u>*poly*</u> r = $(\rho_{-m}, ---, \rho_{-2}, \rho_{-1}, \rho_0, \rho_1, ---, \rho_{n-1}, \rho_n)$

<u>21b</u> <u>*deriv*</u> r = $(-m\times\rho_{-m}, (1-m)\times\rho_{1-m}, ---, -\rho_{-1}, 0, \rho_1, 2\times\rho_2, ---, n\times\rho_n)$

The dyadic form of <u>*deriv*</u>, for example k <u>*deriv*</u> r, returns the kth derivative of its <u>*poly*</u> - i.e. the kth iteration of <u>*deriv*</u> r. You can be sure that the algorithm of the dyadic <u>*deriv*</u> returns *zerovec* without any iteration if $k\geq$ degree of the <u>*poly*</u> (and the <u>*poly*</u> actually represents a polynomial). Also in many other cases <u>*deriv*</u> gives a better performance than iteration could ever do.

3.3 LEVEL2

TORRIX BASIS (chapter 5) consists of the LEVEL1 operations together with those labelled as LEVEL2 (5.0.8, 5.9 and 5.15). Although they are interesting and useful in many practical situations, the essence of level2 is a programming strategy rather than a specific facility.

By declaring *vec*- and *mat*-variables (i.e. *ref vec*- and *ref mat*-identifiers), we free ourselves from irrelevant worries concerning the *array*-bounds. It is worth recognizing that this, in fact, means that we free ourselves from the typical level1-compulsion of having to distinguish the concrete- from the total-*array*s. Another way of saying this is that both kinds of *torrix* variability discussed in 3.1.4, can be managed on level2. We shall see, moreover, that the making of copies can often be better controlled on level2.

There is nothing against the mixed use of *torrix*- and *ref torrix*-identifiers in one program - one might wish, for example, to play off their pros and cons against each other. Normally, however, and certainly in the beginning, it is advisable to stick to one level - which one may then depend on the application-area. We shall, in this section, assume that all vector- and matrix-identifiers are of *ref torrix* mode only, and you should compare their use here with the corresponding examples in 3.2. Compare also the following points with those at the beginning of section 3.2.

When programming TORRIX entirely on level2, all identifiers being of mode *ref vec*, *ref mat* or *ref index*, the essentials of the game are:

- in all assignations to a *ref torrix* destination, the *array*-bounds play no role and no *array* will be copied unless explicitly specified in the source (through the use of the operator *copy*),
- the standard way of holding a newly generated *array*, is by assigning its *vec*, *index* or *mat* to a (level2-)identifier,
- *array*s which are not any more referred to will (thereby, and automatically) be wiped out from the memory (become a willing prey for garbage-collection).

All actions on level2 presuppose the availability of a built-in garbage collector of some quality. Poor garbage collection would imply that many of the nice features of TORRIX LEVEL2 loose their point. At this point we can say that the availability of a garbage collector is an almost formal matter for level1-operations. On level1 it was for syntactic reasons only that we had to generate all *array*s on the heap, and (if we stick to level1) the heap will function as a kind of stack-on-top-of-the-stack (i.e. that heap can be implemented on the stack). Precisely this becomes different on level2 (see also 2.3.2).

Finally we remark that all level1 operators - though requiring *torrix* parameters - accept, without any difficulty *ref torrix* actual-parameters. These will then be dereferenced once - a timeless operation which we can ignore.

3.3.1
The declaration of level2-objects

5.0.2

The general form of a level2-declaration is:

loc torrix z

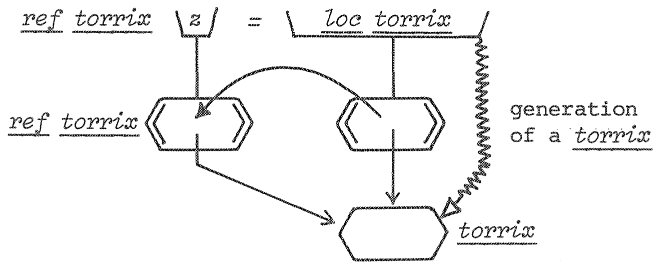
As we have seen in 3.1.5, we shall always write the redundant symbol "*loc*" in order to make a clear distinction with the level1-declarations. In other programming languages one would write perhaps something like "*torrix var z*" as opposed to "*torrix const z*", or "*variable torrix z*", or still better (if the language existed): "*varsize torrix z*". In ALGOL68 the symbol "*loc*" serves that purpose, though it can be omitted.

At this point it becomes interesting to know that in the ALGOL68-orthodoxy "*loc torrix z*" is equivalent to:

ref torrix z = loc torrix

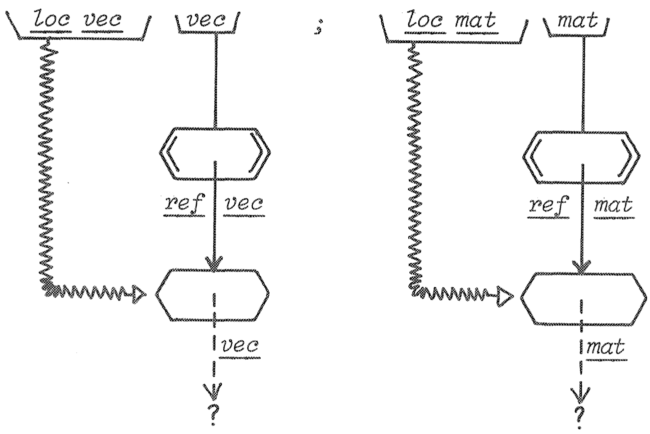
which, more explicitly, states that the identifier *z* is of *ref torrix* mode and that a memory-location for a new *torrix* is being generated:

1



The important feature of level2-declarations, of course, is that they do not generate an *array*:

2



The newly generated *vec* and *mat* have not yet been initialized - they do not yet refer to any *array*. This has been indicated by the question mark in figure 2. The "value" of the question mark is implementation dependent. The formal way of saying is, that it is undefined to which *array* the newly generated *vec* or *mat* will refer.

In the next section we shall see how such new *torrix*es can be initialized (i.e. made to refer to a well-shaped *array*).

Now consider the following sample declarations, which come in the place of those in 3.2.2 and 3.2.5:

D1 *loc vec u,v,w, vec,vec1,vec2 ;*

D2 *loc mat a,b,c, mat,mat1,mat2 ;*

D3 *loc index p,q ;*

In 3.2 (where we stayed at level1) the sizes of *u*, *v*, ... *a*, *b*, ... *p* etc. were fixed at their declaration - their *array*s were generated as a constituent action of that declaration. Level1-declarations were rather drastic, space-reserving actions.

Level2-declarations, to the contrary, are quite harmless. They represent small, purely administrative and not really space-reserving actions. D1, D2 and D3 above define the meaning of the identifiers *u*,*v*,*w*,*vec*,*vec1* and *vec2* to refer to *vec*s, of *a*,*b*,*c*,*mat*,*mat1* and *mat2* ro refer to *mat*s, and of *p* and *q* to refer to *index*es - and nothing more. In particular, they do not fix anything concerning the bounds of these *torrix*es.

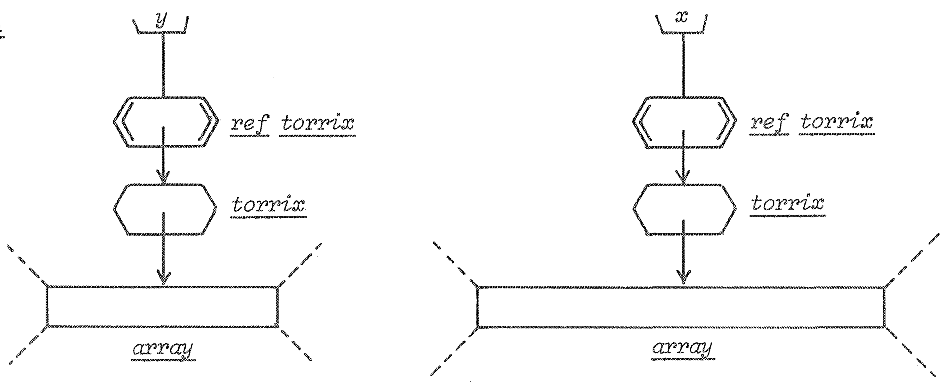
As to their storage allocation, the variable-declarations D1, D2 and D3 above are very well comparable to declarations such as *loc* *int* *n*, *loc* *real* *r*, *loc* *compl* *z*. These also leave the values of *n*, *r* and *z* undefined until initialization. They will never require sudden vast amounts of storage - their memory claim is always modest: just a tiny little *vec* or *mat* as a reference to an *array*, and never more.

On level2 we have complete separation of declaration and *array*-generation.

3.3.2
Level2 assignation

The schemes below depict the situation before and after a level2 assignation:

1a

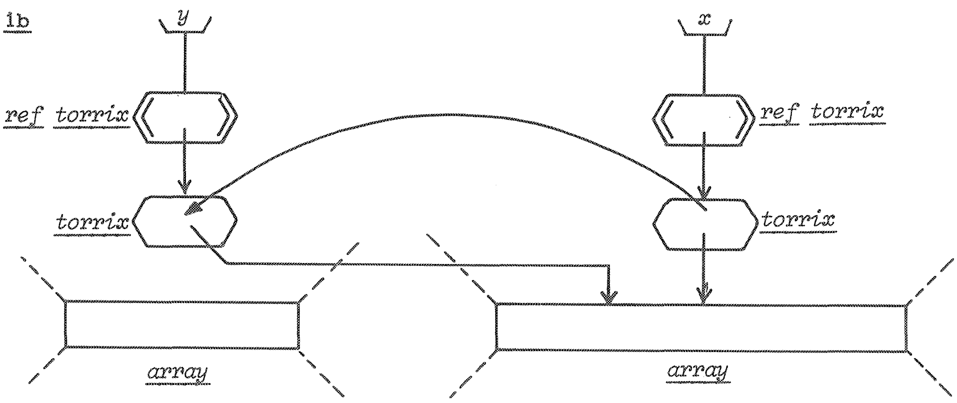


We now assign:

2 $y := x$

By this assignation the value copied ("transported") from "the right to the left" is the *torrix* and not its *array*. Hence, after the assignation, we have:

1b



The *torrix* of *y* ceases to refer to the left *array* because it is made to refer to the right one. Observe that we got a situation in which both *x* and *y* have a depth-reference to the same *array*. The essential point in level2 assignations is that such references can easily be set and reset. There is no *array*-transport involved. All *array*s stay what and where they are. Level2 assignations turn the references and nothing more.

As to what happens to the original *array* of *y* depends on whether some other *torrix* is still interested in it. If not, then that *array* ceases to exist, waiting for the garbage collector. Observe that, by the assignation:

3 $x := z$

the *torrix* of *x* is made to refer to the *array* of *z*; i.e. by 3, *x* ceases to refer to its original *array*. Here then, we have a situation in which the left *array* will not disappear from the memory: if 3 comes after 2, then we still have *y* which is interested in that former *array* of *x*.

We now compare

4a $vec1 := a[i,]$

with 3.2.2.3b. In neither of the two we have *array*-transport.

The difference between them, of course, is that after 3.2.2.3b-LEVEL1 the identifier *rowi* is made to refer to *a*[*i*,] - a relation which is permanent, and assignment to *rowi* means assignment to *a*[*i*,]. After 4a, however, the depth-reference of *vec1* to *a*[*i*,] is temporary and can easily be altered by assigning another *vec* to *vec1*, in which case then *a*[*i*,] remains untouched.

Suppose we also have:

4b *vec2 := a*[*j*,]

We now consider the effect of:

5 *w:=vec1; vec1:=vec2; vec2:=w*

It will be immediately clear that, through 5, *vec1* got a depth-reference to *a*[*j*,], and *vec2* to *a*[*i*,]. Nothing happened to *a*.

For actually exchanging the rows of *a* we have the exchange-operator *=:=* and we write: *a*[*i*,]*=:=a*[*j*,]. Observe that the same effect is achieved by:

6 *vec1=:=vec2*

because *=:=*, being a level1-operator, dereferences *vec1* and *vec2*, and now their *arrays* will actually be touched.

It should be clear without further discussion that the level2 exchange-operation 5 is considerably less drastic (and less time-consuming) than the level1 operation *=:=*.

Where D1, D2 and D3 in this section do not reserve any space for *arrays*, we have to use the generating *gen*-procedures for that purpose. For example:

7 *u := genvec(m); v := genvec(n);*
vec := genarray1(h,k)

8 *a := genmat(m,n); b := genmat(n,k);*
mat := genarray2(h1,k1,h2,k2)

By 7 and 8 we generated 3 *array1*s and 3 *array2*s. The depth-references established by these assignations, however, will last no longer than until another assignation will rearrange them.

Of course it is possible (even recommendable) to initialize *vec*s and *mat*s at their declaration. For example:

9 *loc vec vec1 := genvec(n) , vec2 := zerovec;*
loc mat mat1 := gensquare(n), mat2 := zeromat

The net-effect of

10 $u := v$

is nothing more than that u got the same depth-reference as v. You may, however, wish to get a copy of v and make u refer to that copy. The way to achieve such is:

11 $u :=$ *copy* v

You should carefully compare the essential difference in the result of 10 and 11:

10	$u := v$	u is made to refer to the *array* of v
11	$u :=$ *copy* v	u is made to refer to a new copy of the *array* of v

Interesting is the effect of assignations such as:

12 $u := u[h:k]$ or $u := u[h:k$ *at* $h]$ or $u := u?(h//k)$

After 12, u will not anymore be referring to the "entire" *array*, but to a new descriptor - made by $[h:k]$, $[h:k$ *at* $h]$ or $(h//k)$ respectively - describing the required subvalue of that original *array*. What happens to the dead ends u[1:h-1] and u[k+1:m] depends on whether somevec else is still interested. If not, then these remainders will be swept away (assuming you have got a good garbage-collector). Compare also 3.2.5.9.

Here is an anthology of level2 expressions:

D4 *loc vec* $row, row1, row2, col, col1, col2, diag$;

D5 *loc mat* au, av ;

13	$w := u-v$	no constraints on the bounds;
14	$c := a+b$	no constraints on the bounds;
15	$col := c \times w$	no constraints on the bounds.
16	$col := (a+b) \times (u-v)$	is equivalent to: 13 ; 14 ; 15 apart from automatic removal of intermediate results.

17	*row* := *w*×*c*	no constraints on the bounds.
18	*row* := (*u*−*v*)×(*a*+*b*)	is equivalent to: 13 ; 14 ; 17 apart from automatic removal of intermediate results.
19	*diag* := *diag a*	*diag* gets a depth-reference to the main diagonal of *a*.
20	*diag* := *vec*	*diag* gets a depth-reference to the *array* of *vec* and looses its interest in *diag a*.

If, after 19, you want to assign new values to the main diagonal of *a*, you have to use a level1-assignation on *diag*:

21	*diag*[] := *vec*	this is a level1-assignation, it is equivalent, after 19, to: *diag a* := *vec* the bounds of *diag a* and *vec* have to match.
22	*row1* := *a*[*i*,]	no copy of *a*[*i*,];
23	*row2* := *a*[*j*,]	no copy of *a*[*j*,].
24	*row1* := *row1*−*s*×*row2*	is, after 22 and 23, equivalent to: *row1* := *a*[*i*,]−*s*×*a*[*j*,].

Observe that 24 does not alter *a*[*i*,]. We can even say that 22, 23 and 24 do not change anything in *a* - two of its rows have been referred to. If you want to change *a*[*i*,], you can write the level1-assignation:

25	*row1*[] := *row1*−*s*×*row2*	is, after 22 and 23, equivalent to: *a*[*i*,] := *a*[*i*,]−*s*×*a*[*j*,].
26	*w* := *u* + *v*[*at m*+1]	the source generates the concatenation of *u* and *v* (assuming *upb u* = *m*) and makes *w* refer to that new *array*.
27	*au* := *a* + (*n*+1)*col u*	*au* is made to refer to a new *array2* which contains *a* together with an extra column *u* (assuming 2 *upb a* = *n*).

28 *av := a + (m+1)row v* — *av* is made to refer to a new *array2* which contains *a* together with an extra row *v* (assuming *1 upb a = m*).

Level2 operations, and in particular level2 assignations, play their key role in the manipulation of sparse matrices such as, for example, triangular matrices. For particulars, see TORRIX II. Here is an example:

29 *loc[1:n]vec triang*

which declares (and generates) a row of n *vec*s.

You should verify that the mode of *triang* is *ref[]vec* and, consequently, the (a priori) mode of *triang[i]* is *ref vec*. Hence, if used as the destination in a level2 assignation, *triang[i]* requires from its source a *vec*. This is why we can construct a triangular matrix through the loop-clause:

30 *for i to n do triang[i]:=genvec(i) od*

Finally a few examples of the level2 use of *vec*s which represent *poly*s (compare 3.2.10):

31 *loc poly p,q,r* — *p*, *q* and *r* are *ref ref array1*s representing polynomials or rational functions.

32 *r := p o q* — *r* gets a depth-reference to the (newly generated) polynomial composition of *p* and *q* (compare 3.2.10.18).

33 *r := p××q* — *r* gets a depth-reference to the polynomial (Cauchy-)product of *p* and *q* (compare also 3.2.10.16).

34 *r := (p+q)××(p−q)* — is, ideally and only if the underlying field or ring is commutative, equivalent to: *r := p××2−q××2*

In all these examples we meet a more or less compound formula in the source of a level2 assignation. Intermediate results showing up at the elaboration of these formulae exist no longer than they are needed.

For example in

16 *col := (a+b)×(u−v)*

the intermediate matrixsum $a+b$ and vectordifference $u-v$ will be passed to the operation × and they will cease to exist as soon as the product-computation $(a+b)\times(u-v)$ has been completed. The final result survives for no other reason than that by 16 we made a depth-reference to it through the *ref vec* identifier *col*.

Sizes and *array*-bounds are no concern of ours - the operators and assignments will take care of everything. Moreover, as we have seen, the lifetime of all objects generated during any computation is automatically taken care of by the operations and is further controlled by our own assignations.

3.3.3 Destination-selectors

5.0.8

In 3.1.6 we have seen how the selections *i?u* and *u?i* extend the meaning of *u[i]* beyond its concrete bounds by returning *0* instead of an error-message. Accordingly we have *u?(h//k)* and *(h//k)?u* extending *u[h:k at h]*, *a?i* extending *a[i,]*, *j?a* extending *a[,j]* and *a?(i?j)*, *(i?j)?a*, *i?a?j* etc., all extending *a[i,j]*.

These source-selectors, however, do not actually concretize the virtual elements required. They return the value *0* or a descriptor of a (subvalue of a) concrete *array*. They do not do anything to the concrete-*array* itself. We can write, for instance, *s:=u?i* or *s:=j?a?i*, but never *u?i:=s* nor *j?a?i:=s*. The source-selector *?* returns a *scal* and not a *ref scal* (a virtual zero has no address). With a *trimmer* we may indeed write, for instance, *u?(h//k):=somevec* - provided that *lwb somevec = h max lwb u* and *upb somevec = k min upb u*. However, although *u?(h//k)* returns a *vec* (i.e. a *ref array1*), we remained within the concrete bounds given by the *array1* of *u*.

For some applications - in particular in volume II - we may want to actually extend the given concrete *array* as to comprise the element(s) selected. To that purpose we have the so-called destination-selector *!*.

The important (but often also nasty) side-effect of *!* is, that it immediately generates another concrete-*array* to replace the given one in case it was not "long" enough. As a consequence, *!* must act on a *ref torrix*.

The essence of a destination-selector is, that it validates its corresponding slice, which might have been undefined before the application of the destination-selector:

1	*u!i*	validates	*u*[*i*]	for all *i*
2	*u!(h//k)*	validates	*u*[*h*:*k* at *h*]	for all *h* and *k*
3	*i!u*	validates	*u*[*i*]	for all *i*
4	*(h//k)!u*	validates	*u*[*h*:*k* at *h*]	for all *h* and *k*
5a	*a!(i!j)*	validates	*a*[*i*, *j*]	for all *i* and *j*
		etc.		

Observe that *a?i* returns *zerovec* if *i* is out-of-bounds, but *a!i* has not been defined (difficulties with *a!i* if *a* is *zeromat*).

In order to avoid strange mixes of *!* and *?* we defined *(i!j)* to return the same pair as *(i?j)* so that we can write:

5b *a!(i!j)* instead of *a!(i?j)*

The nasty property of *!* is that it can not know which references exist to (subvalues of) its original concrete-*array*. Consequently, if *!* generates a new concrete-*array*, its *ref torrix* operand gets a depth-reference to that neonate one, but all the possibly existing other *ref*s continue to refer to the obsolete one. The exclamation mark is also a warning.

The reader is in fact mildly advised against *!*, unless he is very sure of his ground - especially of all his underground *ref*s.

An example of an absolute safe, but even so silly, application is:

7 *loc vec new := zerovec; new!(h//k)*

For valuable applications of *!* we refer to volume II. For a few occasional applications see 6.15.1, 6.15.2 and 6.18.5.

3.3.4 Trimming operations

5.9

The operators *trim* and *trims* both require a *ref vec* for their right operand and return a *ref vec*. These trimming operations serve to delimit the shortest possible concrete *array* equivalent (in some sense) to the given one. Through the monadic operator *trim* the equivalence is the strict TORRIX-equivalence, according to the operator = (see 3.2.3 and 5.4.2). The dyadic operator *trims* deals with equivalence in a more numerical sense.

Suppose, for example after an assignation such as *u:=v-w*, we have reason to suspect *u* to contain several "almost zeroes", in particular at the ends of its concrete *array*. We consider a *scal* value *s* to be "almost zero" with respect to a norm *eps≥0*, when *abs s ≤ eps*.

Now the result of:

1 *eps trims u*

can be described in three steps:

- all concrete elements of *u* for which *abs u[i]≤eps* will be set to *0*,
- a new descriptor *[h:k at h]* will be made such that *u?i=0* for all *i<h* and *i>k* but *u[h]≠0* and *u[k]≠0*,
- the sub-*array* with this new descriptor will, through a depth-reference, be assigned to the *ref vec* right operand *u*.

It is important to observe that the "dead ends" will disappear by garbage collection, unless another reference still implies these cut off parts. These parts, however, consist then of concrete zeroes only.

In a sense *trim* and *trims* are the contracting counterparts of *!*. The fundamental difference in practical use, however, is that *trim* and *trims* do not generate a new concrete-*array*. Consequently they are absolutely safe with respect to *ref*s which possibly refer to dead ends.

You should now carefully study the following examples:

2
```
loc[1:m]vec rows ;
for i to m do rows[i]:=a[i, ] od ;
for i to m do eps trims rows[i] od
```

We assigned, with a depth-reference, the rows of a matrix *a* to the *ref vec* elements of *rows* (cf. 3.3.2.29/30). Thereafter we *eps*-trimmed these *rows*[*i*] one after the other. The result is that *rows* now holds the shortest possible concrete *array1*s which are *eps*-equivalent to the rows of *a*. The matrix *a* has also been fashioned - all its almost zeroes have been turned into concrete zeroes. The matrix *a* survives including, of course, its dead ends.

However, when we now assign for example:

3 $\quad$ *a := zeromat*

then the matrix *a* survives only in its trimmed version *rows*. For further manipulations of this kind compare volume II.

The operator *trim* is the purist version of *trims*. The operation *trim u* is equivalent in its result to *(widen 0)trims u*, although it does it with much more efficiency.

In a more mathematical manner we can define the function of *trim* as follows:

For all *vecs u* we have, that after

4 $\quad$ *v:=u;* $\quad$ *trim u*

it is always *true* that

$\qquad (u=v)$ and $(u[\mathit{lwb}\ u]\neq 0$ and $u[\mathit{upb}\ u]\neq 0)$

A certain care in the use of trimming operations is recommendable. Observe that

5 $\quad$ *w := u-v;* $\quad$ *eps trims w*

is another operation than:

6 $\quad$ *w := (eps trims u) - (eps trims v)*

The trimming festival:

7 $\quad$ *w := (eps trims u) - (eps trims v);* $\quad$ *eps trims w*

is almost certainly overdoing it.

In general, one should beware of trimming too much.

A built-in trim after each *array* generating operation would, of course, be fine from a storage-management point of view. The CPU time price, however, may be high.

3.3.5
Level2 assigning additions $\boxed{5.15}$

In 3.2.7 we discussed the level1-assigning operations +<, −<, /> and /<, ×< ×>. The latter three are defined for all feasible *vec* and *mat* operands. In the former three, however, the right operand had to fit in the left operand. On level2 we find ourselves relieved of such constraints.

Expressions such as *u:=u−v* , *a:=a+b* , *u:=u+v[at m+1]* etc. occur frequently in programs. They are all of the form:

$$x := x \pm \text{something}$$

and, therefore, they are obvious candidates for optimization.

In the level2 spirit we do not like bound confinements on one of the operands. This was inevitable on level1 - on level2 we can afford unrestricted operations.

The operations *+:=* and *−:=* fulfill these requirements:

$$\left.\begin{array}{l} x{+}{:=}y \\ x{-}{:=}y \end{array}\right\} \text{ is equivalent to } \left\{\begin{array}{l} x:=x+y \\ x:=x-y \end{array}\right.$$

What happens can be described as:

x+:=y is equivalent to *if y fitsin x then x+<y else x:=x+y fi*
x−:=y is equivalent to *if y fitsin x then x−<y else x:=x−y fi*

It will be clear that, in principle, you should always write *x+:=y* instead of *x:=x+y* and *x−:=y* for *x:=x−y*. The administrative overhead in case *y* does not fit in *x*, is neglectable as compared to the addition/subtraction itself and the gain is considerable when *y fitsin x*.

Pay some attention to the following examples:

1	*u−:=v*	no constraints on the bounds, equivalent to *u:=u−v*.
2	*eps trims(u−:=v)*	as 1; a fashioning operation on *u*, however, follows.
3	*u−:=trim v*	which, under circumstances, might be a good idea.

4 *u+:=v[at upb u +1]* *u* is extended with *v* (compare 3.2.8.7 and 3.3.2.26).

The application 4 of *+:=* is not better (even a tiny little bit slower) than the formulation with *:=* and *+*. It is, however, cleaner. For example:

5 *for i to n do u+:=triang[i][at upb u +1] od*

You should know (or discover) why *triang[i at upb u +1]* would be false in the above example (even syntactically). This example is meant to demonstrate a nice formulation rather than to recommend it. In most implementations it will be more efficient, first to generate the *array1* which can contain all the *triang[i]* and then to assign these to the proper slices.

We can also augment matrices with a column or a row:

6 *a +:= (2 upb a +1)col u* augmentation with a column
7 *a +:= (1 upb a +1)row v* augmentation with a row

Finally, you should verify the following statement:

8 *a +:= i row v* is equivalent to *a[i,]+<v*

in case *v fitsin a[i,]*. However, 8 works also when *not(v fitsin a[i,])*. It is even so, that the overhead of the level2 operation 8, compared to its level1 half-equivalent, is neglectable. So that, provided the operators *row* and *col* have been implemented (see 5.7) in your system, we can even recommend:

9 *a +:= i row v*

and

10 *a +:= j col u*

for the purpose. They are safer than their level1 equivalents.

Of course, it is senseless to prefer *a +:= i row a[j,]*. This one will always result in *a[i,]+<a[j,]*.

4. ORGANIZATIONAL MATTERS

4.1 Notation and terminology 130

4.2 The TORRIX68-message system 133

4.3 Preparatory declarations 137

4.1
Notation and terminology

In the short descriptions of chapters 4 and 5 we use certain technical terms and notations in a more or less fixed meaning - aberrations will always be clear from context. Moreover, for several identifiers we reserved a specific mode - of course, this applies to the description only: in routine texts any identifier may get a different meaning (although we have tried to avoid this).

In the following we list the fixed meaning of notation and terminology in the descriptive (right-)pages:

"total-*array*" "total-*array1*" "total-*array2*"	are the "total-*arrays*" as defined and discussed in chapter 1, expectably containing a specific (may be empty) concrete *array*. The ("virtual") bounds of a total-*array* are *mindex* and *maxdex* and it always consists of a huge number of not-stored "virtual zeroes", together with a relatively small number of potential non-zeroes in its concrete sub-*array*.
"*array*" "*array1*" "*array2*"	are the concrete (sub-)arrays stored in the computer memory and generated by calls of *genarray1* or *genarray2*; the concrete bounds of an *array* are yielded by the operators *lwb* and *upb*, their sizes by the operator *size*.
"*intarray*"	is a concrete []*int* used to store indexes of *arrays*, in order to keep track of permutations of *array*-elements, -rows and -columns.
"*h*", "*i*", "*j*", "*m*" and "*n*"	are *ints* or *int*-variables (*int* or *ref* *int*).
"*k*"	is an *int*, an *int*-variable, a *pair* or a *pair*-variable (*int* or *ref* *int* or *pair* or *ref* *pair*).
"*lwb*", "*upb*"	are *ints* or *int*-variables (*int* or *ref* *int*), denoting the lowerbound or upperbound of a *vec*, *covec* or *index*.
"*r*" and "*s*"	are *scals* or *scal*-variables (*scal* or *ref* *scal*).

"*cr*" and "*cs*"	are *coscal*s or *coscal*-variables	
	(*coscal* or *ref* *coscal*).	
"*p*" and "*q*"	are *index*es, i.e. *ref*[]*intarray*s.	
"*u*", "*v*" and "*w*"	are either *vector*s or *vector*-variables	
	or *covector*s or *covector*-variables	
	(*vec* or *ref* *vec*, *covec* or *ref* *covec*).	
"*a*", "*b*" and "*c*"	are either *matrix*es or *matrix*-variables	
	or *comatrix*es or *comatrix*-variables	
	(*mat* or *ref* *mat*, *comat* or *ref* *comat*).	
"*x*", "*y*" and "*z*"	denote torrixes, i.e.:	
	are either *vector*s or *vector*-variables	
	or *matrix*es or *matrix*-variables	
	or *covector*s or *covector*-variables	
	or *comatrix*es or *comatrix*-variables	
	(*vec* or *ref* *vec* , *mat* or *ref* *mat*,	
	covec or *ref* *covec*, *comat* or *ref* *comat*).	
"*f*"	is a procedure (*proc*).	

4.2

The TORRIX68-message system

```
1.  heap file errorfile;
    int length errorfile = C an int denoting the maximal line length of the
                                book of errorfile
                              C;
    establish(errorfile, "errors", standback channel, 1,1,length errorfile
           );
    loc bool errorfile is open := true;

2.  bool warning = false, fatal = true;
     loc int †numberwarnings := 0;
    proc number of warnings        = int:  †numberwarnings;
    proc reset number of warnings = void:
        (†numberwarnings:=0; reset(errorfile));

3.  proc copyerrorfile = void:
          if errorfile is open
        then putbin(errorfile, -maxint); reset(errorfile);
             loc int line length;
             print((newpage, "torrix errorfile:", newline, newline));
           while getbin(errorfile, line length);
                 line length /= -maxint
              do  if line length < 1
                then print(newline)
                else loc [1 : line length]char line;
                     getbin(errorfile, line);
                     print(("    ", line, newline))
                  fi
              od;
             print((newline, newline, "end torrix errorfile.",
                    newline, newline, newline));
             reset number of warnings
          fi;
```

4.2
The TORRIX68-message system

1. The mode of *errorfile* is <u>*ref*</u> <u>*file*</u>. The book of *errorfile* will contain all warnings, *putbin* writes them down.

The user may close, lock or scratch *errorfile* himself, but then he should not forget to assign <u>*false*</u> to *errorfile is open*. No message will then be sent to *errorfile* and a call of *copyerrorfile* will have no effect.

2. The <u>*bool*</u>-identifiers *warning* and *fatal* serve the readability of the calls of *torrix* in this prelude as also in a users program.

The number of warnings in the book of *errorfile* will be counted by *numberwarnings*. The user can find its value through the procedure *number of warnings* and, moreover, he can reset its value to *0* through the procedure *reset number of warnings*.

A call of *copyerrorfile* performs a call of *reset number of warnings*.

3. A call of *copyerrorfile* copies the book of *errorfile* to *standout* (provided that *errorfile is open* = <u>*true*</u>; if not, nothing happens). This procedure expects a special form of *errorfile*: its book has to consist of a number of lines, where each line contains an integer *n* followed by *n* characters. Each line (except this integer) will be put to a new line of *standout*. A blank line can be put by sending a non-positive integer (not *-maxint*) to *errorfile*.

In the exceptional case that the book of *errorfile* becomes full, the user has to call *copyerrorfile*, not forgetting that *copyerrorfile* expects enough space for writing *-maxint* to the book of *errorfile*.

```
4. proc torrix = (bool fatalerror, []char message)void:
        if fatalerror
       then copyerrorfile; scratch(errorfile);
            errorfile is open := false;
            print((newpage, "fatal error:", newline, message,
                   newline, newline, "trace back:", newline));
            # where possible a trace back #
            goto stop
       elif errorfile is open
       then  numberwarnings +:= 1;
             if upb message max 0 + 65 >  length errorfile -
                                          char number(errorfile)
            then copyerrorfile # assuming an int or char requires each #
             fi;               # 1 position of errorfile #
            putbin (errorfile,
                    (62, "warning! position of standout: page" +
                     whole(page number(standout), -4) + ", line" +
                     whole(line number(standout), -3) + " and char" +
                     whole(char number(standout), -4) + ".",
                     upb message, message, 0)
                   )
        fi;

5. proc stringparam2 = (int n,m)[]char:
       whole(n,0) + " and " + whole(m,0);

   proc stringparam4 = (int k,l,m,n)[]char:
       whole(k,0) + ", " + whole(l,0) + ", " + stringparam2(m,n);

   proc stringindexbounds = (index p)[]char:
       "[" + whole(lwb p,0) + ":" + whole(upb p,0) + "]";

   proc stringvecbounds = (vec u)[]char:
       "[" + whole(lwb u,0) + ":" + whole(upb u,0) + "]";

   proc stringmatbounds = (mat a)[]char:
       "[" + whole(1 lwb a, 0) + ":" + whole(1 upb a, 0) + ", " +
       whole(2 lwb a, 0) + ":" + whole(2 upb a, 0) + "]";

6. # TORRIX68-postlude #
   C stop: copyerrorfile; scratch(errorfile); skip C
```

4. The procedure *torrix* handles errormessages which may be warnings or fatal-errors. In the former case the actual-parameter is *warning*, in the latter case it is *fatal*.

In case of a fatal-error, *copyerrorfile* will be called. It will then scratch *errorfile*, except when *errorfile* has already been scratched, closed or locked. The actual parameter *message* of *torrix* will be sent to *standout*. A traceback will be given (where possible). The program will be terminated.

When a traceback feature has not been implemented, things may happen in a different order.

A call of *torrix* with actual-parameter *warning* and with *errorfile is open* = true counts this event in *numberwarnings*. The current position of *standout* will be sent to the book of *errorfile*. The actual parameter of *message* will be sent to the book of *errorfile*. The integer *0* will be sent to the book of *errorfile*.

When there is not enough space for a warning in the book of *errorfile*, it will be cleared by a call of *copyerrorfile*.

5. A call of the procedures *stringparam2* and *stringparam4* turns the actual parameters into a string, similar actions are performed by *stringindexbounds*, *stringvecbounds* and *stringmatbounds*.

These five procedures and the messages *text1*, *text2*, ..., *text23* (see 6.0) have been declared to shorten the actual parameters in calls of *torrix*.

6. After completion or termination (in case of a fatal-error) of the program, the TORRIX68-postlude prints the errorfile and scratches it.

4.3

Preparatory declarations

```
1.  prio max = 7, min = 7;

       op max = (int m,n)int:
          if m>n then m else n fi;

       op min = (int m,n)int:
          if m<n then m else n fi;

2.  int maxdex = C an int constant such that
                   0 ≤ maxdex ≤ maxint over 2 C;
    int mindex = -maxdex;

    loc int †maxgendex:=maxdex,
            †mingendex:=mindex;

    proc setgendex = (int lower,upper)void:
          if mindex<=lower and upper<=maxdex
        then (mingendex:=lower, maxgendex:=upper);
             torrix(warning, "setgendex: mingendex:="
                           + whole(lower,0)
                           + "maxgendex:=" + whole(upper,0)
                   )
        else torrix(fatal,   "setgendex with forbidden bounds:"
                           + " upper=" + whole(upper,0)
                           + " lower=" + whole(lower,0)
                   )
          fi;

    proc genallowance = (bool yes)void:
          if yes
        then setgendex(mindex,maxdex)
        else setgendex(maxdex,mindex)
          fi;
```

4.3
Preparatory declarations

1. The operators *max* and *min* for *int*-operands are used to find the concrete bounds for operations on *array*s.

2. The virtual bounds *mindex* and *maxdex* are the (implementation-dependent) virtual lowerbound and virtual upperbound of all *array*s (see 1). Consequently, for all concrete *array*s in a program no lowerbound can be less than *mindex* and no upperbound can be greater than *maxdex*.

The condition $maxdex \leq maxint$ *over* 2 is essential, because in some routines the value $maxdex - mindex$ may be computed and should not lead to overflow. The condition $mindex = -maxdex$ is essential for the definition of the reverse inproduct ><.

In many implementations one may find

$$maxdex = 2^n - 1$$

in which n = number of bits in the address-part of an instruction or some other suitable machine-bound integer#.

The generation bounds *maxgendex* and *mingendex* (which are hidden from the user), delimit the index-domain within which *array*s can be generated. The default-values are *maxdex* and *mindex* respectively.

The procedure *setgendex* serves to set particular values for *mingendex* and *maxgendex*. Each call, moreover, results in a warning, reporting which values have been assigned to the generation bounds.

The procedure *genallowance* serves to enable or disable the generation of concrete *array*s, according to the *bool*-value of *yes*. A call *genallowance(true)* resets the generation-bounds *mingendex* and *maxgendex* on their default-values.

On the CYBER/ALGOL68 implementation (CDC-Holland) n=30; for certain optimizations n=18 will be a better choice.

```
3.  mode scal  = # ---, short,   , long, --- # real
              # or rational, or rat
                or, for example, primod (integer modulo prime),
                or any other mode appropriately representing
                or approximating a field or ring or any other
                algebraic system over which vector spaces,
                polynomials, modules etc. can be usefully
                defined
              # ;

    op widen = (int n)scal: C n C
              # the widening from int to scal, which is
                an automatic transfer in case
                mode scal = real
              # ;
```

3. A *scal*, quite generally, may be any mode for which the basic algebraic operations: addition (+), subtraction (−), multiplication (×) and division (/) have been defined in their usual mathematical meaning. A *scal* thus may be any appropriate computer-representation (or -approximation as is the case with *real*, *long real* etc.) for the elements of a field in the mathematical sense.

Particular *scal*-fields may be, for instance:

a) the real-number system **R**, as represented (approximated) by *real*, *long real* etc. or by some other (home-made) mode;

b) the (possibly truncated) field **Q** of rational numbers, as represented (perhaps partially) by a mode *rational*.

c) any finite field, for example Z_p (in which p is a primenumber), as represented by a mode *primod*.

For specific applications - in which division plays no role - the mode *scal* may also represent a ring, for example: *mode scal* = *int* and many other possibilities. For *scal*-rings the corresponding vector-spaces are known as modules.

It will be tacitly assumed that - for all choices of *scal* - the mode *int* is - or can be turned into - a subset of *scal*. Consequently, the *int*-denotations are available to denote certain *scal*-values, in particular zero (*0*) and one (*1*).

One must, however, be aware of the fact that automatic widening exists only in the transfer from *int* to *real* (as also from *real* to *compl*). In the TORRIX68-system most operations necessary to freely use *ints* as specific *scals* will be provided. In the assignation of an *int* to a *scal*-variable, however, the assumption that *int* $\subseteq$ *scal* fails when for *scal* another mode than *real* has been chosen.

```
4.  mode coscal = struct(scal re,im);

      op widen = (scal x)coscal: C x C
                    # the "widening" from scal to coscal,
                      which is automatic when the underlying
                      scal-field is derived from real
                    #;
```

```
5.  prio =:= = 1;

      op =:= = (ref int m,n)ref int:
         (int mn=n; n:=m; m:=mn);

      op =:= = (ref scal r,s)ref scal:
         (scal rs=s; s:=r; r:=rs);

      op =:= = (ref coscal cr,cs)ref coscal:
         (coscal crs=cs; cs:=cr; cr:=crs);
```

4. A *coscal*-field (or -ring) is the complexification of the underlying *scal*-field (or -ring). In case *mode* *scal* = *real* we have: *mode* *coscal* = *compl* and correspondingly for *short*- and *long*-versions.

It is assumed that the specific *coscal*-library (which is standard for *mode* *scal* = ---, *short* *real*, *real*, *long* *real*, ---) provides (apart from the operations +, −, × and /) also the operations:

re , *im* , +× and *conj*.

For the operator *widen*, see 4.3.3.

5. The exchange-operators =:= will be obvious candidates for optimization.

5. TORRIX BASIS

5.0	Fundamental declarations	144
5.1	Array generating procedures	150
5.2	Array generating operations	150
5.3	Bound interrogations	152
5.4	Value interrogations	154
5.5	New values	156
5.6	Straight exchanges	156
5.7	New descriptors only	158
5.8	New descriptors with copies	160
5.9	Trimming operations	160
5.10	Summation and total extrema	162
5.11	Concrete extrema	162
5.12	Level1 assigning additions	164
5.13	Level1 assigning multiplications	166
5.14	Array generating additions	168
5.15	Level2 assigning additions	168
5.16	Array generating multiplications with scalar	170
5.17	Sumproducts	170
5.18	Array generating multiplications	172

5. TORRIX BASIS

5.0

Fundamental declarations | LEVEL 0

```
1.  mode intarray = [mindex:maxdex]int;
    mode array1   = [mindex:maxdex]scal;
    mode array2   = [mindex:maxdex,
                     mindex:maxdex]scal;
```

LEVEL 1

```
2.  mode index    = ref intarray           # ref[]int     #;
    mode vec      = ref array1             # ref[]scal    #;
    mode mat      = ref array2             # ref[,]scal   #;

    vec zerovec   = heap[maxdex:mindex]scal;
    mat zeromat   = heap[maxdex:mindex,
                         maxdex:mindex]scal;
```

5.0

Fundamental declarations

3.1.4/3.1.6/3.3.3

1. The array-modes (*intarray*, *array1* and *array2*) are never explicitly used in TORRIX and any attempt to apply them (as an actual-declarer) will result in an operating system abort - "memory exhausted".
As a formal-declarer, however, they play a role behind the screens. They represent the single- or double-subscripted multiple values referred to by *index*es, *vec*s and *mat*s.
The concrete parts of *array*s can be generated directly by calling the procedures *genarray1*, *genarray2*, *genvec*, *genmat*, *gensquare* etc. and indirectly by applying *array* generating operators (5.14, 5.16 and 5.18).

2. The modes *vec* and *mat* (and *index*) together with the underlying *scal* (and *int*) are the basic-modes of TORRIX. In the ALGOL68-implementation of TORRIX a *vec* (or *mat* or *index*) is nothing more than the name (*ref*) of a concrete *array1* (or *array2* or *intarray*). Nevertheless a *vec* or *mat* as such connotes all information usually attributed to the mathematical concept of a "vector" or a "matrix". See also 1 and 2.

The constants *zerovec* and *zeromat* refer to "empty" concrete *array*s (ultra-flat descriptors) - hence the corresponding total *array*s "contain" virtual zeroes only. Observe that, although assignation is syntactically correct, the only value assignable to *zerovec* (*zeromat*) is *zerovec* (*zeromat*) - as it should be.

It is important and even essential to understand clearly the result of level2-variable-declarations such as:

> *loc vec u,v,w;*   *loc index p,q;*
> *loc mat a,b,c*

To each *u, v, w* (*p, q, a, b, c*) the name ("address") of a newly created *vec* (*index* or *mat*) is ascribed: *u*, *v* and *w* are *vec*-variables (*p* and *q* are *index*-variables, *a*, *b* and *c* are *mat*-variables). Their modes are *ref vec*, *ref index* and *ref mat*.

These declarations do not generate *array*s. To achieve this, one must generate them explicitly.

5.0

Fundamental declarations (continued)

LEVEL 0

3. *mode pair* = *struct(int rowsub,colsub);*
mode trimmer = *struct(int lower ,upper);*

LEVEL 0

	operator	prio	left operand	right operand	result
4	*?* *!*	*9*	*int*	*int*	*pair*
5	*//*	*5*	*int*	*int*	*trimmer*

LEVEL 1

	operator	prio	left operand	right operand	result
6	*fitsin*	*5*	*int* *trimmer* *pair*	*vec* *vec* *mat*	*bool* *bool* *bool*

5.0

Fundamental declarations (continued)

The implementation of the total selectors "∘" and "⫽" (see 1.3.3) meets with difficulties in ALGOL68. The selection of a "slice" from a concrete array is a built-in feature of the language and can not be extended to total arrays (see 2.3.5). The only way around is the declaration of total selectors as operators in ALGOL68. In the nature of things this solution can not be as efficient as a built-in feature. Nevertheless, the selectors may be useable in certain situations, even in TORRIX BASIS (see 6.9.2, 6.15.1/2 and 6.18.5).

For the selector "∘" (subscriptor) we need two ALGOL68-operators: "*?*" (to obtain a current concrete or virtual value) and "*!*" (to change the value selected). For the selector "⫽" (trimmer) we define an ALGOL68-equivalent "//". We also need two selector-modes: *pair* and *trimmer* representing the total equivalents of [*i*, *j*] and [*h*:*k* *at* *h*] respectively.

3.	*pair*((*i*,*j*))	yields the total equivalent of [*i*,*j*] ;
	trimmer((*h*,*k*))	yields the total equivalent of [*h*:*k* *at* *h*].
4.	*i?j* and *i!j*	return *pair*((*i*,*j*)) .
5.	*h//k*	returns *trimmer*((*h*,*k*)).
6.	*i* *fitsin* *u* (*h//k*)*fitsin* *u* (*i?j*)*fitsin* *a* (*i!j*)*fitsin* *a*	when these expressions (are known to) return *true*, we better write *u*[*i*], *u*[*h*:*k* *at* *h*] and *a*[*i*,*j*] for *u?i*, *u?*(*h//k*), *a?*(*i?j*) etc. because no virtual elements are then involved.

5.0

Fundamental declarations (continued)

LEVEL 1

	operator	prio	left operand	right operand	result
7	*?*	*9*	vec	int	scal
			int	vec	scal
			vec	trimmer	vec
			trimmer	vec	vec
			mat	pair	scal
			pair	mat	scal
			mat	int	vec
			int	mat	vec
			mat	trimmer	mat
			trimmer	mat	mat

LEVEL 2

	operator	prio	left operand	right operand	result
8	*!*	*9*	ref vec	int	ref scal
			int	ref vec	ref scal
			ref vec	trimmer	vec
			trimmer	ref vec	vec
			ref mat	pair	ref scal
			pair	ref mat	ref scal

5.0

<u>Fundamental</u> <u>declarations</u> (continued)

7/8. The source-selector "?" and the destination-selector "!" both implement the TORRIX-selector "∘" (see 1.3.3 and 2.3.5). The important ALGOL68-difference between the two, however, is that "!" generates the *array* required if necessary, whereas "?" just returns concrete zeroes where virtual zeroes were selected.

total source-selection	total destination selection	TORRIX-selection	concrete equivalent
u?i	*u!i*	u∘i	*u[i]*
i?u	*i!u*	i∘u	*u[i]*
u?(h//k)	*u!(h//k)*	u∘(h//k)	*u[h:k <u>at</u> h]*
(h//k)?u	*(h//k)!u*	(h//k)∘u	*u[h:k <u>at</u> h]*
a?(i?j)	*a!(i!j)*	a∘i∘j	*a[i,j]*
(i?j)?a	*(i!j)!a*	i∘j∘a	*a[i,j]*
a?i		a∘i	*a[i,]*
j?a		j∘a	*a[,j]*
a?(h//k)		a∘(h//k)	*a[h:k <u>at</u> h,]*
(h//k)?a		(h//k)∘a	*a[,h:k <u>at</u> h]*

NB. Observe that "?" is associative and cyclic, as "∘" is.
For example:
a?(i?j)=(a?i)?j=(j?a)?i=j?(a?i)=(i?j)?a=a?(i?j)=
a?i?j = j?a?i = i?j?a

5.1

Array generating procedures

LEVEL 1

	procedure identifier	1st param.	2nd param.	3rd param.	4th param.	result
1	*genintarray*	int	int			index
2	*genarray1*	int	int			vec
3	*genarray2*	int	int	int	int	mat
4	*genindex*	int				index
5	*genvec*	int				vec
6	*genmat*	int	int			mat
7	*gensquare*	int				mat

5.2

Array generating operations

LEVEL 1

	operator	prio	left operand	right operand	result
1	copy	10		index vec mat	index vec mat
2	span	8	vec mat	vec mat	vec mat
3	meet	8	vec mat	vec mat	vec mat
4	inspan	8	vec mat	vec mat	vec mat
5	subscr	10		vec	index
6	subscr	8	int	mat	index

5.1
<u>Array generating procedures</u> [3.2.2]

1/3. *genintarray(lwb,upb)*, *genarray1(lwb,upb)* and *genarray2(lwb1,upb1,lwb2, upb2)* generate <u>*array*</u>s with the given concrete bounds - they return the <u>*index*</u>, <u>*vec*</u> or <u>*mat*</u> referring to the newly generated <u>*array*</u>;
zerovec or *zeromat* will be returned when a lowerbound is greater than its corresponding upperbound; except for *genintarray*, in which case a flat descriptor with bounds *lwb* and *upb* will be returned;
any violation of the condition †*mingendex* ≤ bound ≤ †*maxgendex* by one of the actual bounds leads to a program-abort.

4. *genindex(size)* is equivalent to *genintarray(1,size)*;
5. *genvec(size)* is equivalent to *genarray1(1, size)*;
6. *genmat(m,n)* is equivalent to *genarray2(1,m,1,n)*;
7. *gensquare(n)* is equivalent to *genarray2(1,n,1,n)*.

5.2
<u>Array generating operations</u> [3.2.2]

1. <u>*copy*</u> *x* — generates a copy of the <u>*array*</u> of *x*.

2. *x* <u>*span*</u> *y* — generates a concrete (zero-)<u>*array*</u> the lowerbounds (upperbounds) of which are the minima (maxima) of the lowerbounds (upperbounds) of *x* and *y*.

3. *x* <u>*meet*</u> *y* — generates a concrete (zero-)<u>*array*</u> the lowerbounds (upperbounds) of which are the maxima (minima) of the lowerbounds (upperbounds) of *x* and *y*.

4. *x* <u>*inspan*</u> *y* — generates a concrete array *x* <u>*span*</u> *y* and assigns the <u>*array*</u> of *x* to *(x* <u>*span*</u> *y)*[<u>*lwb*</u> *x* : <u>*upb*</u> *x*].

5. <u>*subscr*</u> *u* — generates an <u>*intarray*</u> such that (when ascribed or assigned to *p*) <u>*lwb*</u> *p* = <u>*lwb*</u> *u*, <u>*upb*</u> *p* = <u>*upb*</u> *u* and for all <u>*lwb*</u> *u* ≤ *i* ≤ <u>*upb*</u> *u* we have *u*[*p*[*i*]] <u>*is*</u> *u*[*i*].

6. *k* <u>*subscr*</u> *a* — is equivalent to, if *k=1* then <u>*subscr*</u> *a*[,2 <u>*lwb*</u> *a*]
if *k=2* then <u>*subscr*</u> *a*[1 <u>*lwb*</u> *a*,]
i.e. the <u>*subscr*</u> for the columns or rows of *a*;
in case of a flat descriptor in *a*, the <u>*index*</u> returned contains a corresponding flat descriptor.

5.3

Bound interrogations

LEVEL 1

	operator	prio	left operand	right operand	result
1	*lwb*	10		*vec*	*int*
2	*upb*			*index*	*int*
3	*size*			*mat*	*int*
4	*lwb*	8	*int*	*mat*	*int*
5	*upb*				
6	*size*				
7	*fitsin*	5	*index*	*vec*	*bool*
			vec	*vec*	*bool*
			mat	*mat*	*bool*
8	*square*	10		*mat*	*bool*

5.3

Bound interrogations

[3.2.3]

1.	*lwb* x	returns the first concrete lowerbound of x.
2.	*upb* x	returns the first concrete upperbound of x.
3.	*size* x	returns the first concrete size (number of elements) of x.
4.	k *lwb* a	returns the k-lowerbound of a } {k=1 for rows
5.	k *upb* a	returns the k-upperbound of a } {k=2 for columns.
6.	k *size* a	returns the k-size of a {1 *size* a col-size, 2 *size* a row-size.
7.	x *fitsin* y	returns *true* when all bounds of x could be subscripts of y.
8.	*square* a	returns *true* when 1 *size* $a = 2$ *size* a *and* 1 *lwb* $a = 2$ *lwb* a i.e. when a is "square" and centered around the main diagonal.

5.4

Value interrogations

LEVEL 1

	operator	prio	left operand	right operand	result
1	*zero*	*10*		*vec* *mat*	*bool* *bool*
2	= /=	*4*	*vec*	*vec*	*bool*
3	*equ*	*4*	*index*	*index*	*bool*
4	*compat*	*5*	*index*	*vec*	*bool*
5	*search*	*4*	*int*	*index*	*int*

5.4

Value interrogations

3.2.3

1. *zero* x — returns *true* when x *is* *zerovec* or x *is* *zeromat*, or the bounds of x coincide with those of *zerovec* or *zeromat*.

   NB. when *zero* x returns *true* then *size* x (and k *size* x) returns 0; the converse, however, does not always hold.

2. $u = v$ — returns *true* when for all *lwb* u *max* *lwb* $v \leq i \leq$ *upb* u *min* *upb* v we have $u[i] = v[i]$ and, moreover, all other (concrete or virtual) elements are zero; (returns *true* when all elements of the total-*array*s of u and v are equal);

   u /= v — is equivalent to *not*$(u=v)$.

3. p *equ* q — returns *true* when the bounds of p and q are equal and for all *lwb* $p \leq i \leq$ *upb* p we have $p[i] = q[i]$.

4. p *compat* u — returns *true* when p *fitsin* u, and for all *lwb* $p \leq i \leq$ *upb* p we have *lwb* $u \leq p[i] \leq$ *upb* u.

5. k *search* p — returns the smallest subscript i such that $p[i]=k$; the non-existence of such a subscript will be considered as a fatal error.

5.5

New values

LEVEL 1

	operator	prio	left operand	right operand	result
1	*into*	2	*int* *scal* *int* *scal*	*vec* *vec* *mat* *mat*	*vec* *vec* *mat* *mat*
2	*into*	2	*proc(int)scal* *proc(int,int)scal*	*vec* *mat*	*vec* *mat*
3	*into*	2	*int*	*index*	*index*
4	*into*	2	*proc(int)int*	*index*	*index*
5	*identy*	2 10	*int*	*mat* *mat*	*mat* *mat*

5.6

Straight exchanges

LEVEL 1

	operator	prio	left operand	right operand	result
1	=:=	1	*vec*	*vec*	*vec*
2	=:=	1	*index*	*index*	*index*

5.5

New values $\boxed{3.2.5}$

1. *k into x* / *s into x* — assigns the value *k* or *s* to all elements of the *array* of *x*.

2. *f into x* — assigns to each element of the *array* of *x* the corresponding value of *f*, i.e.: $u[i] := f(i)$ or $a[i,j] := f(i,j)$ for all applicable *i* or (i,j).

3. *k into p* — assigns the value of *k* to all elements of the *array* of *p*.

4. *f into p* — assigns to each element of the *array* of *p* the corresponding value of *f*, i.e.: $p[i] := f(i)$ for all applicable *i*.

5. *k identy a* — is equivalent to *(0 into a; 1 into (k diag a));*
   identy a — is equivalent to *0 identy a*, i.e. *identy* returns a "unit-matrix".

5.6

Straight exchanges $\boxed{3.2.5}$

1. *u=:=v* — exchanges the *arrays* of *u* and *v* and returns *u*; the bounds of *u* and *v* have to match.

2. *p=:=q* — exchanges the *intarrays* of *p* and *q* and returns *p*; the bounds of *p* and *q* have to match.

5.7

New descriptors only

LEVEL 1

	operator	prio	left operand	right operand	result
1	*trnsp* *	10		*mat*	*mat*
2	*diag* *	8 10	*int*	*mat* *mat*	*vec* *vec*
3	*col* *	8 10	*int*	*vec* *vec*	*mat* *mat*
4	*row*	8 10	*int*	*vec* *vec*	*mat* *mat*

The operators marked with * are not expressible in ALGOL68 proper, although it must be possible to implement them on all compilers of the full language.

NB. The operator *row* can be expressed in ALGOL68.

5.7

New descriptors only

3.2.5

1. *trnsp* a — constructs - without making a copy of the *array*-elements - a new *array2*-descriptor, so that:
   (*trnsp* a)$[j,i]$ *is* $a[i,j]$ for all applicable (i,j).

2. k *diag* a — constructs - without making a copy of the *array*-elements - a new *array1*-descriptor, so that:
   (k *diag* a)$[i]$ *is* $a[i,i+k]$ for all applicable $(i,i+k)$;

   diag a — is equivalent to 0 *diag* a.

   NB. k *diag* a is defined for all k in the total domain: if the diagonal falls outside the concrete *array2* of a, then the return value is *zerovec*.

3. k *col* u — constructs - without making a copy of the *array*-elements - a new *array2*-descriptor
   [*lwb* u : *upb* u, $k{:}k$], so that:
   (k *col* u)$[i,k]$ *is* $u[i]$ for all applicable i;

   col u — is equivalent to 1 *col* u.

4. k *row* u — constructs - without making a copy of the *array*-elements - a new *array2*-descriptor
   [$k{:}k$, *lwb* u : *upb* u], so that:
   (k *row* u)$[k,i]$ *is* $u[i]$ for all applicable i;

   row u — is equivalent to 1 *row* u.

NB. *col* and *row* both present a *vec* as if it were a *mat* (with one column or row) without copying the *array* referred to.

5.8

New descriptors with copies

LEVEL 1

	operator	prio	left operand	right operand	result
1	*copytrnsp*	10		*mat*	*mat*
2	*copydiag*	8 10	*int*	*mat* *mat*	*vec* *vec*
3	*copycol*	8 10	*int*	*vec* *vec*	*mat* *mat*
4	*copyrow*	8 10	*int*	*vec* *vec*	*mat* *mat*

5.9

Trimming operations

LEVEL 2

	operator	prio	left operand	right operand	result
1	*trims*	8	*scal*	*ref vec*	*ref vec*
2	*trim*	10		*ref vec*	*ref vec*

5.8

New descriptors with copies

3.2.5

1. *copytrnsp a* is equivalent to *copy trnsp a*.
2. *k copydiag a* is equivalent to *copy(k diag a)*;
   copydiag a is equivalent to *copy diag a*.
3. *k copycol u* is equivalent to *copy(k col u)*;
   copycol u is equivalent to *copy col u*.
4. *k copyrow u* is equivalent to *copy(k row u)*;
   copyrow u is equivalent to *copy row u*.

5.9

Trimming operations

3.3.4

1. *eps trims u* — assigns *0* to all concrete elements of *u* with absolute value ≤*eps* and constructs a new descriptor for that *array* in order to achieve that:
   (u[lwb u]/=0)and(u[upb u]/=0);
   i.e. "*trims*" fashions its operand into the shortest possible concrete *array1*.
2. *trim u* — constructs a new descriptor for that *array* in order to achieve that:
   (u[lwb u]/=0)and(u[upb u]/=0);
   i.e. "*trim*" fashions its operand into the shortest possible concrete *array1*.

NB. The result of *trims* or *trim* may be the assignment of *zerovec* to *u*, in which case the above wordings must be rephrased accordingly.

5.10

Summation and total extrema

LEVEL 1

	operator	prio	left operand	right operand	result
1	*sigma*	10		*vec* *mat*	*scal* *scal*
2	*sigmabs*	10		*vec* *mat*	*scal* *scal*
3	*max*	10		*vec* *mat*	*scal* *scal*
4	*min*	10		*vec* *mat*	*scal* *scal*
5	*maxabs*	10		*vec* *mat*	*scal* *scal*
6	*minabs*	10		*vec* *mat*	*scal* *scal*

5.11

Concrete extrema

LEVEL 1

	operator	prio	left operand	right operand	result
1 2 3 4	*max* *min* *maxabs* *minabs*	7	*ref int* *ref pair*	*vec* *mat*	*scal* *scal*
5 6	*max* *min*	7	*ref int*	*index*	*int*

5.10

Summation and total extrema

3.2.6

1. *sigma* x — returns the sum of all elements of the total-*array* of x.

2. *sigmabs* x — returns the sum of the absolute values of all elements of the total-*array* of x.

3. *max* x — returns the value of the maximal element of the total-*array* of x (inclusive virtual zeroes).

4. *min* x — returns the value of the minimal element of the total-*array* of x (inclusive virtual zeroes).

5. *maxabs* x — same as 3, but now for *abs*-values.

6. *minabs* x — same as 4, but now for *abs*-values (NB. *minabs* $x = 0$ for all x).

5.11

Concrete extrema

3.2.6

1. k *max* x — returns the value of the maximal element of the concrete *array* of x and assigns its (smallest) subscript(s) to k; a program-abort follows when *size* $x = 0$.

2. k *min* x — returns the value of the minimal element of the concrete *array* of x and assigns its (smallest) subscript(s) to k; a program-abort follows when *size* $x = 0$.

3. k *maxabs* x — same as 1, but now for *abs*-values.

4. k *minabs* x — same as 2, but now for *abs*-values.

5. k *max* p — returns the maximal element of the *intarray* of p and assigns its (smallest) subscript to k;
a program-abort follows when *size* $p = 0$.

6. k *min* p — returns the minimal element of the *intarray* of p and assigns its (smallest) subscript to k;
a program-abort follows when *size* $p = 0$.

NB. The operations 5.10 apply to the total *array*s (inclusive the virtual zeroes), whereas the operations 5.11 apply to the concrete *array*s only (exclusive the virtual zeroes).

5.12

Level1 assigning additions

LEVEL 1

	operator	prio	left operand	right operand	result
1	+<	*1*	<u>*index*</u> <u>*vec*</u> <u>*mat*</u>	<u>*int*</u> <u>*scal*</u> <u>*scal*</u>	<u>*index*</u> <u>*vec*</u> <u>*mat*</u>
2	−<	*1*	<u>*index*</u> <u>*vec*</u> <u>*mat*</u>	<u>*int*</u> <u>*scal*</u> <u>*scal*</u>	<u>*index*</u> <u>*vec*</u> <u>*mat*</u>
3	+<	*1*	<u>*vec*</u> <u>*mat*</u>	<u>*vec*</u> <u>*mat*</u>	<u>*vec*</u> <u>*mat*</u>
4	−<	*1*	<u>*vec*</u> <u>*mat*</u>	<u>*vec*</u> <u>*mat*</u>	<u>*vec*</u> <u>*mat*</u>

5.12

Level1 assigning additions

3.2.7

1. *p* +< *k* — adds to all elements of the *intarray* of *p* the *int* *k*;

   x +< *s* — adds to all elements of the concrete *array* of *x* the *scal* *s*.

2. *p* −< *k* — subtracts from all elements of the *intarray* of *p* the *int* *k*;

   x −< *s* — subtracts from all elements of the concrete *array* of *x* the *scal* *s*.

3. *x* +< *y* — adds to the elements of the concrete *array* of *x* the corresponding elements of the concrete *array* of *y*, provided that *y* *fitsin* *x* - violation of this condition results in a program-abort.

4. *x* −< *y* — subtracts from the elements of the concrete *array* of *x* the corresponding elements of the concrete *array* of *y*, provided that *y* *fitsin* *x* - violation of this condition results in a program-abort.

5.13

Level1 assigning multiplications

LEVEL 1

	operator	prio	left operand	right operand	result
1	×<	*1*	*vec*	*int*	*vec*
			vec	*scal*	*vec*
			mat	*int*	*mat*
			mat	*scal*	*mat*
2	×>	*1*	*int*	*vec*	*vec*
			scal	*vec*	*vec*
			int	*mat*	*mat*
			scal	*mat*	*mat*
3	/<	*1*	*vec*	*int*	*vec*
			vec	*scal*	*vec*
			mat	*int*	*mat*
			mat	*scal*	*mat*
4	*neg*	*10*		*vec*	*vec*
5	×>	*1*	*vec*	*vec*	*vec*
6	/>	*1*	*vec*	*vec*	*vec*

5.13

Level1 assigning multiplications

3.2.7

1.	*x* ×< *n*	multiplies, from the right, all elements of the *array* of *x* with *n*;
	x ×< *s*	multiplies, from the right, all elements of the *array* of *x* with *s*.
2.	*n* ×> *x*	multiplies, from the left, all elements of *x* with *n*;
	s ×> *x*	multiplies, from the left, all elements of *x* with *s*.
3.	*x* /< *n*	divides all elements of the *array* of *x* by *n*;
	x /< *s*	divides all elements of the *array* of *x* by *s*.
4.	*neg* *x*	is equivalent to *x* ×< *-1*, but presumably more efficient.
5.	*u* ×> *v*	multiplies each element of the total-*array* of *v* with the corresponding element of the total-*array* of *u*; the return-value is *v*.
6.	*u* /> *v*	divides each element of the concrete *array* of *v* by the corresponding element of the *array* of *u*, provided that *v* *fitsin* *u* - violation of this condition results in a program-abort; the return-value is *v*.

Recommended pronunciation of the level1-assigning arithmetic operators:

+<	"plus from"
−<	"minus from"

×<	"times from"
/<	"divided from"
×>	"times into"
/>	"divides into"

5.14

Array generating additions

LEVEL 1

	operator	prio	left operand	right operand	result
1	+	*6*	***vec*** ***mat***	***vec*** ***mat***	***vec*** ***mat***
2	−	*6*	***vec*** ***mat***	***vec*** ***mat***	***vec*** ***mat***
3	−	*10*		***vec*** ***mat***	***vec*** ***mat***

5.15

Level2 assigning additions

LEVEL 2

	operator	prio	left operand	right operand	result
1	+:=	*1*	***ref vec*** ***ref mat***	***vec*** ***mat***	***ref vec*** ***ref mat***
2	−:=	*1*	***ref vec*** ***ref mat***	***vec*** ***mat***	***ref vec*** ***ref mat***

5.14

Array generating additions

3.2.8

1. $x + y$ — generates x span y and assigns to its elements the sum $\upsilon_i+\phi_i$ or $\alpha_{ij}+\beta_{ij}$.
2. $x - y$ — generates x span y and assigns to its elements the differences $\upsilon_i-\phi_i$ or $\alpha_{ij}-\beta_{ij}$.
3. $-x$ — is equivalent to *zerovec-u* or *zeromat-a*.

5.15

Level2 assigning additions

3.3.5

1. x +:= y — is, in its result, equivalent to x := x+y; however, when y *fitsin* x, the operation x+<y is performed - hence, x+:=y may be considerably more efficient than x := x+y.
2. x −:= y — is, in its result, equivalent to x := x−y; however, when y *fitsin* x, the operation x−<y is performed - hence, x−:=y may be considerably more efficient than x := x−y.

5.16

Array generating multiplications with scalar

LEVEL 1

	operator	prio	left operand	right operand	result
1	×	7	*int*	*vec*	*vec*
			scal	*vec*	*vec*
			int	*mat*	*mat*
			scal	*mat*	*mat*
			vec	*int*	*vec*
			vec	*scal*	*vec*
			mat	*int*	*mat*
			mat	*scal*	*mat*
2	/	7	*vec*	*int*	*vec*
			vec	*scal*	*vec*
			mat	*int*	*mat*
			mat	*scal*	*mat*

5.17

Sumproducts

LEVEL 1

	operator	prio	left operand	right operand	result
1	×	7	*vec*	*vec*	*scal*
2	<>	7	*vec*	*vec*	*scal*
3	><	7	*vec*	*vec*	*scal*
4	*o*	8	*vec*	*int*	*scal*
			vec	*scal*	*scal*

5.16

Array generating multiplications with scalar

3.2.10

1. $n \times x$ is equivalent to $n{\times}{>}(\underline{copy}\ x)$;
   $s \times x$ is equivalent to $s{\times}{>}(\underline{copy}\ x)$;
   $x \times n$ is equivalent to $(\underline{copy}\ x){\times}{<}n$;
   $x \times s$ is equivalent to $(\underline{copy}\ x){\times}{<}s$.
2. x / n is equivalent to $(\underline{copy}\ x)/{<}n$;
   x / s is equivalent to $(\underline{copy}\ x)/{<}s$.

5.17

Sumproducts

3.2.9

1. $u \times v$ returns $\sum \upsilon_i \phi_i$.
2. $u{<>}v$ returns $\sum \upsilon_i \phi_i$.

   NB. The operators $\times$ and $<>$ accomplish in all respects the same. The reason for two different operator-symbols will be found in TORRIX-COMPLEX where the innerproduct $u{<>}v$ is $\sum \upsilon_i \bar{\phi}_i$, whereas $u{\times}v$ remains $\sum \upsilon_i \phi_i$.
3. $u{><}v$ returns $\sum \upsilon_i \phi_{-i}$.

   NB. The "reverse sum-product" $><$ serves to form amongst others the convolution (Cauchy-)product of polynomials.
4. $u\ \underline{o}\ n$
   $u\ \underline{o}\ s$ the so-called "Horner-product" of u - conceived as a <u>polynomial</u> ($\underline{lwb}\ u \geq 0$), or a <u>rational</u> <u>function</u> ($\underline{lwb}\ u < 0$) - for the value n or s;
   i.e. the value of the function u for n or s.

5.18

Array generating multiplications

LEVEL 1

	operator	prio	left operand	right operand	result
1	×	*7*	*mat* *vec* *mat*	*vec* *mat* *mat*	*vec* *vec* *mat*
2 3	*trnspmul* *multrnsp*	*10*		*mat*	*mat*
4	××	*8*	*vec* *vec*	*vec* *int*	*vec* *vec*
5	*o*	*8*	*vec*	*vec*	*vec*
6	*deriv*	*7*	*int*	*vec*	*vec*
7	*deriv*	*10*		*vec*	*vec*

5.18

Array generating multiplications | 3.2.10 |

1. $a \times u$ matrix × "column" returns "column": $\sum \alpha_{ij}\upsilon_j$;
   $u \times a$ "row" × matrix returns "row" : $\sum \upsilon_i \alpha_{ij}$;
   $a \times b$ matrix × matrix returns matrix : $\sum \alpha_{hi}\beta_{ik}$.

2. *trnspmul a*   is equivalent to *trnsp a* × *a*, but is also defined when *trnsp a* is not available.

3. *multrnsp a*   is equivalent to *a* × *trnsp a*, but is also defined when *trnsp a* is not available.

4. $u \times\times v$ the convolution (Cauchy-)product of u and v;

   NB. The *vec*s u and v are conceived as representing polynomials (lwbs ≥ *0*) and/or rational functions (lwbs < *0*). The product $u\times\times v$ then returns a new polynomial or rational function such that, ideally, $(u \; \underline{o} \; s)\times(v \; \underline{o} \; s) = (u\times\times v) \; \underline{o} \; s$;

   $u \times\times n$ the nth "Cauchy power" of u;

   NB. $(u \; \underline{o} \; s)\times\times n = (u\times\times n) \; \underline{o} \; s$, ideally.

5. $u \; \underline{o} \; v$ the composition of u and v;

   NB. u is conceived as representing a polynomial (*lwb* $u \geq 0$) and v is conceived as representing a polynomial or a rational function. Their composition $u \; \underline{o} \; v$ then returns a function for which we have, ideally: $(u \; \underline{o} \; v) \; \underline{o} \; s = u \; \underline{o} \; (v \; \underline{o} \; s)$.

6. *k deriv u*   returns the kth derivative of u (conceived as a polynomial or a rational function); $k \geq 0$, violation of this condition results in a program-abort.

7. *deriv u*   is equivalent to *1 deriv u*.

6. BASIS: ROUTINE-TEXTS

6.0	Fundamental- and hidden operations, messages	176
6.1	Array generating procedures	181
6.2	Array generating operations	183
6.3	Bound interrogations	185
6.4	Value interrogations	186
6.5	New values	188
6.6	Straight exchanges	190
6.7	New descriptors only	191
6.8	New descriptors with copies	192
6.9	Trimming operations	193
6.10	Summation and total extrema	194
6.11	Concrete extrema	196
6.12	Level1 assigning additions	200
6.13	Level1 assigning multiplications	202
6.14	Array generating additions	205
6.15	Level2 assigning additions	206
6.16	Array generating multiplications with scalar	207
6.17	Sumproducts	208
6.18	Array generating multiplications	209

6. BASIS: ROUTINE-TEXTS

6.0

Fundamental- and hidden operations, messages

```
prio †plusab = 6, †minab = 6;

  op †plusab = (vec u, array1 v)vec:
     (for i from lwb v to upb v
       do u[i] +:= v[i] od; u
     );

  op †plusab = (mat a, array2 b)mat:
     (for j from 2 lwb b to 2 upb b
       do a[ ,j] plusab b[ ,j] od; a
     );

  op †minab  = (vec u, array1 v)vec:
     (for i from lwb v to upb v
       do u[i] -:= v[i] od; u
     );

  op †minab  = (mat a, array2 b)mat:
     (for j from 2 lwb b to 2 upb b
       do a[ ,j] minab b[ ,j] od; a
     );
```

NB. The operators †*plusab* and †*minab* are hidden from the user. They accomplish certain routines without feasibility-checks (which are supposed to have been done in the routines using them) they are therefore dangerous for direct use. They are also obvious candidates for essential optimization.

The peculiar second *array*-parameter enforces the making of a copy, presumedly postponed until it becomes absolutely inevitable (namely for the rare cases where the actual *array*s ill-fatedly overlap).

See also 1.3.2 for this problem.

```
4.  op ?  = (int i,j)pair: pair((i,j));

    op !  = (int i,j)pair: pair((i,j));

5.  op // = (int h,k)trimmer: trimmer((h,k));

6.  op fitsin = (int i, vec u)bool:
       i >= lwb u and i <= upb u;

    op fitsin = (trimmer slice, vec u)bool:
       lower of slice >= lwb u and upper of slice <= upb u;

    op fitsin = (pair ij, mat a)bool:
        if rowsub of ij < 1 lwb a or rowsub of ij > 1 upb a
      then false
      else colsub of ij >= 2 lwb a and colsub of ij <= 2 upb a
        fi;

7.  op ? = (vec u, int i)scal:
       if i fitsin u then u[i] else widen 0 fi;

    op ? = (int i, vec u)scal:
       if i fitsin u then u[i] else widen 0 fi;

    op ? = (vec u, trimmer slice)vec:
        if int h = lwb u max lower of slice,
               k = upb u min upper of slice;
           h > k
      then zerovec
      else u[h:k at h]
        fi;

    op ? = (trimmer slice, vec u)vec: u?slice;

    op ? = (mat a, pair ij)scal:
        if ij fitsin a
      then a[rowsub of ij, colsub of ij] else widen 0
        fi;

    op ? = (pair ij, mat a)scal:
        if ij fitsin a
      then a[rowsub of ij, colsub of ij] else widen 0
        fi;
```

```
    op ? = (mat a, int i)vec:
        if i < 1 lwb a or i > 1 upb a then zerovec else a[i, ] fi;

    op ? = (int j, mat a)vec:
        if j < 2 lwb a or j > 2 upb a then zerovec else a[ ,j] fi;

    op ? = (mat a, trimmer slice)mat:
        if int h = lower of slice max 1 lwb a,
               k = upper of slice min 1 upb a;
            h > k
      then zeromat
      else a[h:k at h, ]
        fi;

    op ? = (trimmer slice, mat a)mat:
        if int h = lower of slice max 2 lwb a,
               k = upper of slice min 2 upb a;
            h > k
      then zeromat
      else a[ ,h:k at h]
        fi;

8.  op ! = (ref vec u, int i)ref scal:
       (if not(i fitsin u)
      then int lwb = lwb u, upb = upb u;
            vec v = 0 into genarray1(lwb min i, upb max i);
            v[lwb:upb at lwb] := u; u := v
        fi; u[i]
       );

    op ! = (int i, ref vec u)ref scal: u!i;

    op ! = (ref vec u, trimmer slice)vec:
       (int h = lower of slice, k = upper of slice;
         if not(slice fitsin u)
       then int lwb = lwb u, upb = upb u;
             vec v = 0 into genarray1(h min lwb, k max upb);
             v[lwb:upb at lwb] := u; u := v
         fi; u[h:k at h]
       );
```

```
op ! = (trimmer slice, ref vec u)vec: u!slice;

op ! = (ref mat a, pair ij)ref scal:
   (int i = rowsub of ij, j = colsub of ij;
    if not(ij fitsin a)
   then int la1 = 1 lwb a, ua1 = 1 upb a,
            la2 = 2 lwb a, ua2 = 2 upb a;
        mat b = 0 into genarray2(i min la1, i max ua1,
                                 j min la2, j max ua2);
        b[la1:ua1 at la1, la2:ua2 at la2] := a; a := b
    fi; a[i,j]
   );

op ! = (pair ij, ref mat a)ref scal: a!ij;
```

9.

```
[]char
†text1  = "call of genarray with parameters "                     ,
†text2  = "attempt to generate an array beyond bounds. "          ,
†text3  = "the parameters were: "                                 ,
†text4  = ". result is zerovec."                                  ,
†text5  = ". result is zeromat."                                  ,
†text6  = "failure in using operator search. "                    ,
†text7  = "the value of the left operand was "                    ,
†text8  = " and the bounds of the vector were: "                  ,
†text9  = "bounds in operator =:= for vectors do not match: "     ,
†text10 = "bounds in operator =:= for indexes do not match: "     ,
†text11 = "monadic operator minabs returns always zero."          ,
†text12 = "empty array in dyadic max. bounds of array: "          ,
†text13 = "empty array in dyadic min. bounds of array: "          ,
†text14 = "empty array in dyadic maxabs. bounds of array: "       ,
†text15 = "empty array in dyadic minabs. bounds of array: "       ,
†text16 = "empty index in dyadic max. bounds of index: "          ,
†text17 = "empty index in dyadic min. bounds of index: "          ,
†text18 = "incompatible bounds in operator +<: "                  ,
†text19 = "incompatible bounds in operator -<: "                  ,
†text20 = "right operand of ×× is a negative integer: "           ,
†text21 = ". bounds of vector: "                                  ,
†text22 = "incompatible bounds in operator />: "                  ,
†text23 = "left operand of deriv is a negative integer: "         ;
```

NB. The hidden-status of the texts enables the implementer to store them in the most appropriate manner.

6.1

Array generating procedures

1.
```
proc genintarray = (int lwb,upb)index:
      ( if lwb<mingendex or upb>maxgendex or
             upb<mingendex or lwb>maxgendex
        then torrix(fatal, text2 + text3 + stringparam2(lwb,upb) + ".")
        elif lwb>upb
        then torrix(warning, text1 + stringparam2(lwb,upb) + ".")
          fi; heap[lwb:upb]int
      );
```
2.
```
proc genarray1 = (int lwb,upb)vec:
        if lwb>upb
       then torrix(warning, text1 + stringparam2(lwb,upb) + text4);
            zerovec
       elif lwb<mingendex or upb>maxgendex
       then torrix(fatal, text2 + text3 + stringparam2(lwb,upb) + ".");
            skip
       else heap[lwb:upb]scal
        fi;
```
3.
```
proc genarray2 = (int lwb1, upb1, lwb2, upb2)mat:
        if lwb1>upb1 or lwb2>upb2
       then torrix(warning, text1 + stringparam4(lwb1,upb1,lwb2,upb2)
                   + text5
                  );
            zeromat
       elif lwb1<mingendex or lwb2<mingendex or
            upb1>maxgendex or upb2>maxgendex
       then torrix(fatal, text2 + text3
                   + stringparam4(lwb1,upb1,lwb2,upb2) + "."
                  );
            skip
       else heap[lwb1:upb1, lwb2:upb2]scal
        fi;
```

```
4.  proc genindex  = (int size)index: genintarray(1,size);
5.  proc genvec    = (int size)vec:   genarray1(1, size);
6.  proc genmat    = (int m,n)mat:    genarray2(1,m,1,n);
7.  proc gensquare = (int n)mat:      genarray2(1,n,1,n);
```

6.2

Array generating operations

1.
```
op copy = (index p)index:
    genintarray(lwb p, upb p) := p;

op copy = (vec u)vec:
      if size u = 0
    then zerovec
    else genarray1(lwb u, upb u) := u
      fi;

op copy = (mat a)mat:
      if 1 size a = 0 or 2 size a = 0
    then zeromat
    else genarray2(1 lwb a, 1 upb a, 2 lwb a, 2 upb a) := a
      fi;
```

2.
```
op span = (vec u,v)vec:
    0 into genarray1(lwb u min lwb v, upb u max upb v);

op span = (mat a,b)mat:
    0 into genarray2(1 lwb a min 1 lwb b, 1 upb a max 1 upb b,
                     2 lwb a min 2 lwb b, 2 upb a max 2 upb b);
```

3.
```
op meet = (vec u,v)vec:
    0 into genarray1(lwb u max lwb v, upb u min upb v);

op meet = (mat a,b)mat:
    0 into genarray2(1 lwb a max 1 lwb b, 1 upb a min 1 upb b,
                     2 lwb a max 2 lwb b, 2 upb a min 2 upb b);
```

4.
```
op inspan = (vec u,v)vec:
      if v fitsin u
    then copy u
    else vec w = u span v, int lu = lwb u;
         w[lu: upb u at lu] := u; w
      fi;
```

```
    op inspan = (mat a,b)mat:
         if b fitsin a
       then copy a
       else mat c = a span b, int la1 = 1 lwb a, la2 = 2 lwb a;
             c[la1:.1 upb a at la1, la2: 2 upb a at la2] := a; c
         fi;

5.  op subscr = (vec u)index:
        (int lwb = lwb u, upb = upb u;
        index subscr = genintarray(lwb,upb);
         for i from lwb to upb
          do subscr[i] := i od; subscr
        );

6.  op subscr = (int k, mat a)index:
        (int lwb = k lwb a, upb = k upb a;
        index subscr = genintarray(lwb,upb);
         for i from lwb to upb
          do subscr[i] := i od; subscr
        );
```

6.3

Bound interrogations

```
3.  op size = (vec u)int:
       0 max (upb u - lwb u +1);

    op size = (index p)int:
       0 max (upb p - lwb p +1);

    op size = (mat a)int: 1 size a;

6.  op size = (int k, mat a)int:
       0 max (k upb a - k lwb a +1);


7.  op fitsin = (index p, vec u)bool:
       lwb p >= lwb u and upb p <= upb u;

    op fitsin = (vec u,v)bool:
       lwb u >= lwb v and upb u <= upb v;

    op fitsin = (mat a,b)bool:
       1 lwb a >= 1 lwb b and 1 upb a <= 1 upb b and
       2 lwb a >= 2 lwb b and 2 upb a <= 2 upb b;

8.  op square = (mat a)bool:
       1 size a = 2 size a and 1 lwb a = 2 lwb a;
```

NB. The operators *lwb* and *upb* (TORRIX68 numbers 6.3.1 and 6.3.2) belong to the ALGOL68 'standard prelude'; their defining occurence can be found in 10.2.3.1 of {36}.

6.4
Value interrogations

```
1.  op zero = (vec u)bool:
        if u is zerovec
      then true
      else lwb u = maxdex and upb u = mindex
        fi;

    op zero = (mat a)bool:
        if a is zeromat
      then true
      elif 1 lwb a = maxdex and 1 upb a = mindex
      then 2 lwb a = maxdex and 2 upb a = mindex
      else false
        fi;

2.  op = = (vec u,v)bool
        if loc vec x := u, y := v;
           (trim x, trim y);
           int lwb = lwb x, upb = upb x;
           lwb /= lwb y or upb /= upb y
      then false
      else loc bool result := true;
           for i from lwb to upb
         while result := x[i] = y[i]
            do skip od; result
        fi;

    op /= = (vec u,v)bool: not(u=v);
```

NB. The application of *trim* in 6.4.2 is an optimization. This routinetext must be adapted for TORRIX68 systems which do not support level2.

```
3.  op equ = (index p,q)bool:
        if int lwb = lwb p, upb = upb p;
              lwb = lwb q and upb = upb q
      then loc bool result := true;
             for i from lwb to upb
           while result := p[i] = q[i]
               do skip od; result
      else false
        fi;

4.  op compat = (index p, vec u)bool:
        if p fitsin u
      then loc bool result := true;
             int lwb = lwb u, upb = upb u;
             for i from lwb p to upb p
           while int pi = p[i];
                 result := pi>=lwb and pi<=upb
               do skip od; result
      else false
        fi;

5.  op search = (int k, index p)int:
        if loc bool notthis := true, loc int subscr := lwb p;
               to size p
           while notthis := p[subscr] /= k
               do subscr +:= 1 od; notthis
      then torrix(fatal, text6 + text7 + whole(k,0) + text8
                   + stringindexbounds(p) + "."
                  );
             skip
      else subscr
        fi;
```

6.5

New values

```
1.  op into = (int n, vec u)vec: widen n into u;
              # The performance of this routine may heavily depend #
              # on the scal chosen. The present routine assumes     #
              # scal to be L real.                                  #

    op into = (scal s, vec u)vec:
       (for i from lwb u to upb u
         do u[i] := s od; u
       );

    op into = (int n, mat a)mat: widen n into a;

    op into = (scal s, mat a)mat:
       (for j from 2 lwb a to 2 upb a
         do s into a[ ,j] od; a
       );

2.  op into = (proc(int)scal f, vec u)vec:
       (for i from lwb u to upb u
         do u[i] := f(i) od; u
       );

    op into = (proc(int,int)scal f, mat a)mat:
       (for j from 2 lwb a to 2 upb a
         do vec colj = a[ ,j];
            for i from 1 lwb a to 1 upb a
             do colj[i] := f(i,j) od
         od; a
       );

3.  op into = (int k, index p)index:
       (for i from lwb p to upb p
         do p[i] := k od; p
       );
```

```
4.  op into = (proc(int)int f, index p)index:
       (for i from lwb p to upb p
         do p[i] := f(i) od; p
       );

5.  op identy = (int k, mat a)mat:
       (0 into a; 1 into (k diag a); a);

    op identy = (mat a)mat: 0 identy a;
```

6.6

Straight exchanges

```
1.  op =:= = (vec u,v)vec:
       if int lwb = lwb u, upb = upb u;
          lwb = lwb v and upb = upb v
     then for i from lwb to upb
            do u[i] =:= v[i] od; u
     else torrix(fatal, text9 + stringvecbounds(u) + " and "
                   + stringvecbounds(v) + "."
                  );
          skip
       fi;

2.  op =:= = (index p,q)index:
       if int lwb = lwb p, upb = upb p;
          lwb = lwb q and upb = upb q
     then for i from lwb to upb
            do p[i] =:= q[i] od; p
     else torrix(fatal, text10 + stringindexbounds(p) + " and "
                   + stringindexbounds(q) + "."
                  );
          skip
       fi;
```

NB. The natural applications of these operators are the exchanges of rows or columns of matrices; the *arrays* will then never overlap one another. If such might be the case, an intermediate copy may be inevitable (depending on the overlap and the order of exchange). These routinetexts are obvious candidates for optimization, which should take this problem into account (cf. 1.3.2).

6.7

New descriptors only

1. *op trnsp = (mat a)mat:*
   C a mat such that,
   for all subscripts i and j within the bounds of a:
   (trnsp a)[j,i] is a[i,j]
   C;

2. *op diag = (int k, mat a)vec:*
   C a vec such that,
   for all subscripts i and i+k within the bounds of a:
   (k diag a)[i] is a[i,i+k]
   C;

   op diag = (mat a)vec: 0 diag a;

3. *op col = (int k, vec u)mat: trnsp (k row u);*

   op col = (vec u)mat: trnsp (row u);

4. *op row = (int k, vec u)mat:*
   if zero u then zeromat else mat(u)[at k,] fi;

   op row = (vec u)mat:
   if zero u then zeromat else mat(u) fi;

NB. The operators 6.7.1 and 6.7.2 cannot be defined in ALGOL68 proper (cf. 2.1.4 and 2.3.6). As a consequence, also the operators 6.7.3 (which rely on them) do not belong to TORRIX68*.

6.8

New descriptors with copies

1. <u>*op*</u> <u>*copytrnsp*</u> *= (*<u>*mat*</u> *a)*<u>*mat*</u>*:* <u>*copy*</u> <u>*trnsp*</u> *a;*

2. <u>*op*</u> <u>*copydiag*</u> *= (*<u>*int*</u> *k,* <u>*mat*</u> *a)*<u>*vec*</u>*:* <u>*copy*</u> *(k* <u>*diag*</u> *a);*

   <u>*op*</u> <u>*copydiag*</u> *= (*<u>*mat*</u> *a)*<u>*vec*</u>*:* <u>*copy*</u> <u>*diag*</u> *a;*

3. <u>*op*</u> <u>*copycol*</u> *= (*<u>*int*</u> *k,* <u>*vec*</u> *u)*<u>*mat*</u>*:* <u>*copy*</u> *(k* <u>*col*</u> *u);*

   <u>*op*</u> <u>*copycol*</u> *= (*<u>*vec*</u> *u)*<u>*mat*</u>*:* <u>*copy*</u> <u>*col*</u> *u;*

4. <u>*op*</u> <u>*copyrow*</u> *= (*<u>*int*</u> *k,* <u>*vec*</u> *u)*<u>*mat*</u>*:* <u>*copy*</u> *(k* <u>*row*</u> *u);*

   <u>*op*</u> <u>*copyrow*</u> *= (*<u>*vec*</u> *u)*<u>*mat*</u>*:* <u>*copy*</u> <u>*row*</u> *u;*

6.9

Trimming operations

1.
```
op trims = (scal eps, ref vec u)ref vec:
    (loc int newlwb := lwb u, newupb := upb u;
     int sizu = size u, scal zero = widen 0;
      to sizu
   while ref scal ui = u[newlwb]; abs ui <= eps
      do (newlwb +:= 1, ui := zero) od;
      to newupb-newlwb
   while ref scal ui = u[newupb]; abs ui <= eps
      do (newupb -:= 1, ui := zero) od;
      if newupb<newlwb
    then u := zerovec
    else for i from newlwb+1 to newupb-1
          do if abs u[i] <= eps then u[i] := zero fi
          od;
          u := u[newlwb: newupb at newlwb]
      fi
    );
```

2.
```
op trim = (ref vec u)ref vec:
    (int newlwb := lwb u, newupb := upb u;
     int sizu = size u;
      to sizu while u[newlwb] = 0
      do newlwb +:= 1 od;
      to newupb-newlwb while u[newupb] = 0
      do newupb -:= 1 od;
      u := u?(newlwb//newupb)
    );
```

6.10

Summation and total extrema

1.
```
op sigma = (vec u)scal:
    (loc scal s := widen 0;
     for i from lwb u to upb u
      do s +:= u[i] od; s
    );

op sigma = (mat a)scal:
    (loc scal s := widen 0;
     for j from 2 lwb a to 2 upb a
      do s +:= sigma a[ ,j] od; s
    );
```

2.
```
op sigmabs = (vec u)scal:
    (loc scal s := widen 0;
     for i from lwb u to upb u
      do s +:= abs u[i] od; s
    );

op sigmabs = (mat a)scal:
    (loc scal s := widen 0;
     for j from 2 lwb a to 2 upb a
      do s +:= sigmabs a[ ,j] od; s
    );
```

3.
```
op max = (vec u)scal:
    (loc scal max := widen 0;
     for i from lwb u to upb u
      do if u[i]>max then max := u[i] fi od; max
    );

op max = (mat a)scal:
    (loc scal max := widen 0, submax;
     for j from 2 lwb a to 2 upb a
      do submax := max a[ ,j];
         if submax>max then max := submax fi
      od; max
    );
```

```
4.  op min = (vec u)scal:
       (loc scal min := widen 0;
        for i from lwb u to upb u
         do if u[i]<min then min := u[i] fi od; min
       );

    op min = (mat a)scal:
       (loc scal min := widen 0, submin;
        for j from 2 lwb a to 2 upb a
         do submin := min a[ ,j];
            if submin<min then min := submin fi
         od; min
       );

5.  op maxabs = (vec u)scal:
       (loc scal maxabs := widen 0;
        for i from lwb u to upb u
         do if abs u[i]>maxabs then maxabs := abs u[i] fi
         od; maxabs
       );

    op maxabs = (mat a)scal:
       (loc scal maxabs := widen 0, submax;
        for j from 2 lwb a to 2 upb a
         do submax := maxabs a[ ,j];
            if submax>maxabs then maxabs := submax fi
         od; maxabs
       );

6.  op minabs = (vec u)scal: (torrix(warning, text11); widen 0);

    op minabs = (mat a)scal: (torrix(warning, text11); widen 0);
```

6.11

Concrete extrema

1.
```
op max = (ref int index, vec u)scal:
    if size u = 0
  then torrix(fatal, text12 + stringvecbounds(u) + "."); skip
  else index := lwb u; loc scal max := u[index];
       for i from index+1 to upb u
        do if u[i]>max
         then (max := u[i], index := i)
           fi
        od; max
    fi;

op max = (ref pair ij, mat a)scal:
    if 2 size a = 0
  then torrix(fatal, text12 + stringmatbounds(a) + "."); skip
  else loc int index, loc scal submax,
       max := rowsub of ij max a[ , colsub of ij := 2 lwb a];
       for j from colsub of ij +1 to 2 upb a
        do   if (submax := index max a[ ,j])>max
            then (max := submax, ij := (index,j))
              fi
        od; max
    fi;
```

2.
```
op min = (ref int index, vec u)scal:
    if size u = 0
  then torrix(fatal, text13 + stringvecbounds(u) + "."); skip
  else index := lwb u; loc scal min := u[index];
       for i from index+1 to upb u
        do if u[i]<min
         then (min := u[i], index := i)
           fi
        od; min
    fi;
```

```
op min = (ref pair ij, mat a)scal:
    if 2 size a = 0
  then torrix(fatal, text13 + stringmatbounds(a) + "."); skip
  else loc int index, loc scal submin,
       min := rowsub of ij min a[ , colsub of ij := 2 lwb a];
       for j from colsub of ij +1 to 2 upb a
        do   if (submin := index min a[ ,j])<min
           then (min := submin, ij := (index,j))
             fi
        od; min
    fi;

3. op maxabs = (ref int index, vec u)scal:
    if size u = 0
  then torrix(fatal, text14 + stringvecbounds(u) + "."); skip
  else index := lwb u; loc scal maxabs := abs u[index];
       for i from index+1 to upb u
        do if abs u[i]>maxabs
         then (maxabs := abs u[i], index := i)
           fi
        od; maxabs
    fi;

op maxabs = (ref pair ij, mat a)scal:
    if 2 size a = 0
  then torrix(fatal, text14 + stringmatbounds(a) + "."); skip
  else loc int index, loc scal submax,
       maxabs := rowsub of ij maxabs a[ , colsub of ij := 2 lwb a];
       for j from colsub of ij +1 to 2 upb a
        do   if (submax := index maxabs a[ ,j])>maxabs
           then (maxabs := submax, ij := (index,j))
             fi
        od; maxabs
    fi;
```

```
4.  op minabs = (ref int index, vec u)scal:
        if size u = 0
      then torrix(fatal, text15 + stringvecbounds(u) + "."); skip
      else index := lwb u; loc scal minabs := abs u[index];
           for i from index+1 to upb u
            do if abs u[i]<minabs
             then (minabs := abs u[i], index := i)
               fi
            od; minabs
        fi;

    op minabs = (ref pair ij, mat a)scal:
        if 2 size a = 0
      then torrix(fatal, text15 + stringmatbounds(a) + "."); skip
      else loc int index, loc scal submin,
           minabs := rowsub of ij minabs a[ , colsub of ij := 2 lwb a];
           for j from colsub of ij +1 to 2 upb a
            do   if (submin := index minabs a[ ,j])<minabs
                then (minabs := submin, ij := (index,j))
                  fi
            od; minabs
        fi;

5.  op max = (ref int subscr, index p)int:
        if size p = 0
      then torrix(fatal, text16 + stringindexbounds(p) + "."); skip
      else subscr := lwb p; loc int max := p[subscr];
           for i from subscr+1 to upb p
            do if p[i]>max
             then (max := p[i], subscr := i)
               fi
            od; max
        fi;
```

```
6.  op min = (ref int subscr, index p)int:
        if size p = 0
      then torrix(fatal, text17 + stringindexbounds(p) + "."); skip
      else subscr := lwb p; loc int min := p[subscr];
           for i from subscr+1 to upb p
            do if p[i]<min
             then (min := p[i], subscr := i)
               fi
            od; min
        fi;
```

6.12

Level1 assigning additions

1.
```
op +< = (index p, int k)index:
    (for i from lwb p to upb p
      do p[i] +:= k od; p
    );

op +< = (vec u, scal s)vec:
    (for i from lwb u to upb u
      do u[i] +:= s od; u
    );

op +< = (mat a, scal s)mat:
    (for j from 2 lwb a to 2 upb a
      do a[ ,j] +< s od; a
    );
```

2.
```
op -< = (index p, int k)index:
    (for i from lwb p to upb p
      do p[i] -:= k od; p
    );

op -< = (vec u, scal s)vec:
    (for i from lwb u to upb u
      do u[i] -:= s od; u
    );

op -< = (mat a, scal s)mat:
    (for j from 2 lwb a to 2 upb a
      do a[ ,j] -< s od; a
    );
```

```
3.  op +< = (vec u,v)vec:
       if v fitsin u
      then u plusab v
      else torrix(fatal, text18 + stringvecbounds(u) + " and "
                    . + stringvecbounds(v) + "."
                   );
          skip
       fi;

    op +< = (mat a,b)mat:
       if b fitsin a
      then a plusab b
      else torrix(fatal, text18 + stringmatbounds(a) + " and "
                     + stringmatbounds(b) + "."
                   );
          skip
       fi;

4.  op -< = (vec u,v)vec:
       if v fitsin u
      then u minab b
      else torrix(fatal, text19 + stringvecbounds(u) + " and "
                     + stringvecbounds(v) + "."
                   );
          skip
       fi;

    op -< = (mat a,b)mat:
       if b fitsin a
      then a minab b
      else torrix(fatal, text19 + stringmatbounds(a) + " and "
                     + stringmatbounds(b) + "."
                   );
          skip
       fi;
```

6.13

Level1 assigning multiplications

```
1.  op ×< = (vec u, int n)vec:
        (for i from lwb u to upb u
          do u[i] ×:= n od; u
        );

    op ×< = (vec u, scal s)vec:
        (for i from lwb u to upb u
          do u[i] ×:= s od; u
        );

    op ×< = (mat a, int n)mat:
        (for j from 2 lwb a to 2 upb a
          do a[ ,j] ×< n od; a
        );

    op ×< = (mat a, scal s)mat:
        (for j from 2 lwb a to 2 upb a
          do a[ ,j] ×< s od; a
        );

2.  op ×> = (int n, vec u)vec:
        (for i from lwb u to upb u
          do ref scal ui = u[i]; ui := n×ui
          od; u
        );

    op ×> = (scal s, vec u)vec:
        (for i from lwb u to upb u
          do ref scal ui = u[i]; ui := s×ui
          od; u
        );

    op ×> = (int n, mat a)mat:
        (for j from 2 lwb a to 2 upb a
          do n ×> a[ ,j] od; a
        );
```

```
    op ×> = (scal s, mat a)mat:
        (for j from 2 lwb a to 2 upb a
          do s ×> a[ ,j] od; a
        );

3.  op /< = (vec u, int n)vec:
        (for i from lwb u to upb u
          do u[i] /:= n od; u
        );

    op /< = (vec u, scal s)vec:
        (for i from lwb u to upb u
          do u[i] /:= s od; u
        );

    op /< = (mat a, int n)mat:
        (for j from 2 lwb a to 2 upb a
          do a[ ,j] /< n od; a
        );

    op /< = (mat a, scal s)mat:
        (for j from 2 lwb a to 2 upb a
          do a[ ,j] /< s od; a
        );

4.  op neg = (vec u)vec: u×<-1;

5.  op ×> = (vec u,v)vec:
         if int low = lwb u max lwb v, up = upb u min upb v;
           for i from low to up
             do v[i] ×:= u[i] od;
           v fitsin u
       then v
       elif low>upb v or up<lwb v or low>up
       then 0 into v
       else (0 into v[ :low-1], 0 into v[up+1: ]); v
         fi;
```

```
6.  op /> = (vec u,v)vec:
       if v fitsin u
      then for i from lwb v to upb v
             do v[i] /:= u[i] od; v
      else torrix(fatal, text22 + stringvecbounds(u) + " and "
                     + stringvecbounds(v) + "."
                   );
           skip
       fi;
```

NB. An intermediate copy of *u* may be inevitable in case the concrete *array*s of *u* and *v* ill-fatedly overlap (cf. 1.3.2).

6.14

<u>Array generating additions</u>

1. *<u>op</u> + = (<u>vec</u> u,v)<u>vec</u>: (u <u>inspan</u> v)<u>plusab</u> v;*

   <u>op</u> + = (<u>mat</u> a,b)<u>mat</u>: (a <u>inspan</u> b)<u>plusab</u> b;

2. *<u>op</u> − = (<u>vec</u> u,v)<u>vec</u>: (u <u>inspan</u> v)<u>minab</u> v;*

   <u>op</u> − = (<u>mat</u> a,b)<u>mat</u>: (a <u>inspan</u> b)<u>minab</u> b;

3. *<u>op</u> − = (<u>vec</u> u)<u>vec</u>: <u>neg</u> <u>copy</u> u;*

   <u>op</u> − = (<u>mat</u> a)<u>mat</u>: zeromat−a;

6.15

Level2 assigning additions

```
1.  op +:= = (ref vec u, vec v)ref vec:
        if v fitsin u then u plusab v; u
      else u := u+v
        fi;

      # u!(lwb v // upb v) plusab v #

    op +:= = (ref mat a, mat b)ref mat:
        if b fitsin a then a plusab b; a
      else a := a+b
        fi;

2.  op -:= = (ref vec u, vec v)ref vec:
        if v fitsin u then u minab v; u
      else u := u-v
        fi;

      # u!(lwb v // upb v) minab v #

    op -:= = (ref mat a, mat b)ref mat:
        if b fitsin a then a minab b; a
      else a := a-b
        fi;
```

6.16

Array generating multiplications with scalar

```
1.  op × = (int n, vec u)vec:
        n ×> copy u;

    op × = (scal s, vec u)vec:
        s ×> copy u;

    op × = (int n, mat a)mat:
        n ×> copy a;

    op × = (scal s, mat a)mat:
        s ×> copy a;

    op × = (vec u, int n)vec:
        copy u ×< n;

    op × = (vec u, scal s)vec:
        copy u ×< s;

    op × = (mat a, int n)mat:
        copy a ×< n;

    op × = (mat a, scal s)mat:
        copy a ×< s;

2.  op / = (vec u, int n)vec:
        (copy u) /< n;

    op / = (vec u, scal s)vec:
        (copy u) /< s;

    op / = (mat a, int n)mat:
        (copy a) /< n;

    op / = (mat a, scal s)mat:
        (copy a) /< s;
```

6.17

Sumproducts

```
1.  op × = (vec u,v)scal:
      (loc scal prod := widen 0;
       for i from  lwb u max lwb v  to  upb u min upb v
        do prod +:= u[i]×v[i] od; prod
      );

2.  op <> = (vec u,v)scal: u×v;

3.  op >< = (vec u,v)scal:
      (loc scal revprod := widen 0;
       for i from  lwb u max -upb v  to  upb u min -lwb v
        do revprod +:= u[i]×v[-i] od; revprod
      );

4.  op o = (vec u, int j)scal:
      (loc vec x := u; trim x;
       loc scal us := widen 0, int lwb = lwb x;
       for i from upb x by -1 to lwb
        do (us ×:= j) +:= u[i] od;
        if lwb<0 then us/(j××-lwb) else us×(j××lwb) fi
      );

    op o = (vec u, scal s)scal:
      (loc vec x := u; trim x;
       loc scal us := widen 0, int lwb = lwb x;
       for i from upb x by -1 to lwb
        do (us ×:= s) +:= u[i] od; us × (s××lwb)
      );
```

NB. The application of *trim* in 6.17.4 is an optimization, and can simply be left out in TORRIX68 systems which do not support level2.

6.18

Array generating multiplications

```
1.  op × = (mat a, vec u)vec:
       if 2 upb a < lwb u or upb u < 2 lwb a
      then zerovec
      else int lwb1 = 1 lwb a, upb1 = 1 upb a;
           vec v = genarray1(lwb1,upb1);
           for i from lwb1 to upb1
            do v[i] := a[i, ]×u od; v
       fi;

    op × = (vec u, mat a)vec: (trnsp a) × u;

    op × = (mat a,b)mat:
       if 1 upb b < 2 lwb a or 2 upb a < 1 lwb b
      then zeromat
      else int blwb2 = 2 lwb b, bupb2 = 2 upb b;
           mat c = genarray2(1 lwb a, 1 upb a, blwb2, bupb2);
           for j from blwb2 to bupb2
            do c[ ,j] := a×b[ ,j] od; c
       fi;

2.  op trnspmul = (mat a)mat:
       if zero a
      then zeromat
      else mat at = trnsp a, int lwb2 = 2 lwb a, upb2 = 2 upb a;
           mat ata = genarray2(lwb2,upb2,lwb2,upb2);
           for i from lwb2 to upb2
            do vec atai = ata[i: at i, u];
               atai := at[i: at i, ] × a[ ,i];
               ata[i, i+1: at i+1] := atai[i+1: at i+1]
            od; ata
       fi;
```

```
3.  op multrnsp = (mat a)mat:
       if zero a
      then zeromat
      else int lwb1 = 1 lwb a, upb1 = 1 upb a;
           mat aat = genarray2(lwb1,upb1,lwb1,upb1);
           for i from lwb1 to upb1
            do vec aati = aat[i: at i, i];
               aati := a[i: at i, ] × a[i, ];
               aat[i, i+1: at i+1] := aati[i+1: at i+1]
            od; aat
       fi;

4.  op xx = (vec u,v)vec:
       if zero u or zero v
      then zerovec
      else int lwbv = lwb v;
           vec w = genarray1(lwb u + lwbv, upb u + upb v);
           for i from lwb w to upb w
            do w[i] := u >< v[at lwbv-i] od; w
       fi;
```

```
op ×× = (vec u, int n)vec:
   if n >= 0
  then case n+1
        in 1 into genarray1(0,0), copy u,
           u××u, u××u××u, (vec v = u××u; v××v)
       out vec v = u××u;
            if not odd n
          then v××(n over 2)
          elif n mod 3 /= 0 and n /= 23
          then u ×× (v××(n over 2))
          elif vec w = v××u; n=23
          then w ×× ((v××w)××4)
          else (w××(n over 3)) ×× 3
            fi
      esac
  elif int lwb = lwb u, upb = upb u; lwb = upb
  then int pow = lwb×n;
      (u[lwb]××n) into genarray1(pow,pow)
  else torrix(fatal, text20 + whole(n,0) + text21
             + stringvecbounds(u) + "."
            ); skip
   fi;
```

```
5.  op o = (vec u,v)vec:
        if loc vec x := u; zero trim x
      then zerovec
      elif int lwb = lwb x, upb = upb x;
           loc vec y := v; trim y; vec last = y××lwb; lwb = upb
      then last ×< u[upb]
      elif loc vec uv := u[upb] × y;
           for i from upb-1 by -1 to lwb+1
            do uv!0 +:= u[i]; uv := uv××y od;
           uv!0 +:= u[lwb];
           lwb = 0
      then uv
      else uv××last
        fi;
```

NB. The application of *trim* and *!* in 6.18.5 are optimizations. This routinetext must be adapted for TORRIX68 systems which do not support level2.

```
6.  op deriv = (int k, vec u)vec:
     case k+1
       in copy u, deriv u
      out  if k<0
         then torrix(fatal, text23 + whole(k,0) + text21
                      + stringvecbounds(u) + "."
                    );
              skip
         elif loc int lwb := lwb u, upb := upb u;
              (if lwb>=0 and lwb<k then lwb := k fi,
               if upb>=0 and upb<k then upb := -1 fi
              ); lwb>upb
         then zerovec
         else vec v = 0 into genarray1(lwb-k, upb-k);
              ((loc int exprod := (upb-k+1) max 0;
                for i from exprod+1 to upb
                 do exprod ×:= i od;
                for i from upb-k by -1 to (lwb-k) max 0
                 do v[i] := u[i+k]×exprod;
                    (exprod overab i+k) ×:= i
                 od
               ),
               (loc int exprod := lwb min 0;
                for i from exprod-1 by -1 to lwb-k+1
                 do exprod ×:= i od;
                for i from lwb-k to (upb-k) min -k
                 do v[i] := u[i+k]×exprod;
                    (exprod overab i+1) ×:= i+k+1
                 od
               )
              ); v
           fi
     esac;
```

```
7.  op deriv = (vec u)vec:
        if int lwb = lwb u + abs(lwb u = 0),
               upb = upb u - abs(upb u = 0);
           upb<lwb
      then zerovec
      else vec v = copy u[lwb:upb at lwb-1];
           for i from lwb-1 to upb-1
            do v[i] ×:= i+1 od; v
        fi;
```

INDEX

BIBLIOGRAPHY

INDEX

addition	1.1.1, 1.1.2, 1.1.4,1.2.4,4.3.3
array generating ~	3.2.8, 5.14, 6.14
level1 assigning ~	3.2.7, 5.12, 6.12
level2 assigning ~	3.3.5, 5.15, 6.15
address	see reference
array (*array*)	1.2.2, 2.3, 3.1.3, 3.1.4, 3.2.1, 4.1, 5.0.1
~ bounds	see bounds
concrete ~	1.2.3, 2.1.5, 2.3.1, 3.1.4, 3.1.6, 3.3.1, 4.1
domain of ~	1.2.2, 1.2.4, 1.3.1, 2.3.1
empty ~	1.2.2, 3.1.3, see *zerovec*, *zeromat*
equivalence of ~s	1.2.2, 3.2.3, 5.4.2, 6.4.2
generation of an ~	2.3.3, 3.1.4, 3.1.5, 3.2.2, 3.3.1, 4.1
total ~	1.2.3, all of 2.3, 3.1.4, 3.1.6, 3.2.5, 4.1
array1, array2 (*array1*, *array2*)	1.2.2, 4.1, 5.0.1
ascription	3.1.5
level1 ~	see level1
assignation	2.2.1, 3.1.5, 3.2.5
level1 ~	see level1
level2 ~	see level2
bound	
array (array) ~	3.1.3, 3.1.5, 3.1.6, 3.2.1, 4.1, 4.3
generation ~	2.3.3, 3.2.1, 4.3.2
~ interrogation	3.2.3, 5.3, 6.3
virtual ~	3.2.1, 4.1
cauchy	
~ power	see polynomial
~ product	see polynomial and convolution
column	3.1.4, 3.2.5, 3.2.10, 5.7.3, 6.7.3, 5.8.3, 6.8.3
~ of a matrix	see matrix
compl	2.2.4, 2.3.4, 3.1.2, 4.3.4
complex number (C)	1.1, 1.1.1, 1.2.1, 2.2.4, 4.3.4
concrete	see array, domain, extrema, operation
control construct/structure	2.1.2, 2.1.3, 3.1.1
convolution	3.2.10, 5.18.4, 6.18.4

declaration	
generating level1 ~	3.1.5
identifier ~	3.1.5
identity ~	3.1.5, 3.2.1
initializing ~	3.1.5
level1 ~	see level1
level2 ~	see level2
mode ~	2.1.2, 4.3, 5.0
operation ~	2.2, 3.1.1
procedure ~	3.1.1
declarer	3.1.1, 3.1.2
depth reference	see reference
dereferencing	3.1.5, 3.3
weak ~	3.1.5
derivative	see polynomial
descriptor	2.1.5, 2.3.6, 3.1.3, 3.2.5, 3.3.4
(ultra) flat ~	3.1.3, 3.2.1, 3.2.3
destination	2.2.1, 3.1.5
diagonal of a matrix	see matrix
dimension	1.1.3, 2.1.1
display	
row ~	3.1.1
division	1.1.1, 1.1.2, 4.3, 5.13, 5.16, 6.13, 6.16
rational ~	2.2.3
domain	
~ of an array	see array
concrete ~	1.3.1
index ~	2.2.1, 4.3.2
~ variability	1.3.1, 2.3.1
dyadic operation	see operation
elementwise	see operation
embedded	1.2.3, 3.1.4
empty	
~ array	see array
~ trimmer	3.1.5
equal vectors	3.2.3
equivalence of array's	see array
errorfile	see message
errormessage	see message

euclidean
- ~ space — see space
- ~ vectornorm — 1.1.5, 3.2.9

exchange operator — 3.1.1, 3.2.5, 3.3.2, 4.3.5, 5.6, 6.6

extrema
- concrete ~ — 3.2.6, 5.11, 6.11
- total ~ — 3.2.6, 5.10, 6.10

field — 1.1.1, 3.1.2, 3.2, 4.3.3

finite dimensional — see dimension

flat descriptor — see descriptor

flex, flexible — 2.1.3, 2.3, 3.1.1

generation
- ~ of an array — see array
- ~ bounds — see bounds

generator — 2.3.2, 2.3.3, 3.1.1, 3.1.5

goto — 3.1.1

group — 1.1.1

heap (heap) — see generator

hidden — 2.1.4, 2.1.5, 2.2, 6.0

Hilbert matrix — 1.3, 2.3.7

Horner product — 5.17.4, 6.17.4, 6.18.5

identifier — 3.1.5
- ~ declaration — see declaration
- *ref torrix* ~ — 3.1.5
- *torrix* ~ — 3.1.5

identity
- ~ declaration — see declaration
- ~ transformation — 1.1.2
- ~ operator — 1.1.2, 3.2.5, 5.5, 6.5

index — 2.2.1, 3.1.2
- ~ domain — see domain

indexer — 3.1.3

initializing declaration — see declaration

innerproduct — 1.1.5, 3.2.9, 5.17.2, 6.17.2
- ~ space — see space

int	2.2.1, 4.3.3
int-overflow	see overflow
integral	2.2.3, 3.1.2
integral number (Z)	1.1, 1.1.1, 1.2.1, 2.2.1, 2.2.3, 3.1.2
interrogation	
bound ~	see bounds
value ~	see value
inverse	1.1.1
label	3.1.1
layer	2.2, 2.2.6
level0	2.3.1, 3.1.4, 3.2.1, 5.0.1
level1	2.3.1, 2.3.2, 3.1.4, all of 3.2
~ *array* variable	3.1.5
~ ascription	3.1.5, 3.2.4
~ assignation	3.1.5, 3.2.4
~ assigning addition	see addition
~ assigning multiplication	see multiplication
~ assigning operation	3.2.7, 5.12, 5.13, 6.12, 6.13
~ declaration	3.1.5, 3.2.2
generating ~ declaration	3.1.5
~ object	2.3.1, 3.1.4
~ variability	2.3.1, 3.1.4
level2	2.3.1, 2.3.2, 3.1.4, all of 3.3
~ assignation	3.1.4, 3.1.5, 3.3.2
~ assigning addition	see addition
~ assigning operations	see operation
~ declaration	3.1.5, 3.3.1
~ object	2.1.3, 3.1.4
~ variable	3.1.5, 3.3.1
~ variability	2.3.1, 3.1.4
library	
(*co*)*scal*-~	2.2, 4.3
operation ~	3.1.2, see mode/operator package
~ prelude	see prelude
linear	
~ combination	1.1.3
~ly independent	1.1.3
~ operator	1.1.2
~ transformation	1.1.2, 1.1.4, 2.1.1
composition of ~ transformations	1.1.2, 3.2.9
lowerbound	2.1.5, 3.1.3

matrix	1.1.4, 1.2.4, 2.1.1, see *mat*
column of a ~	1.1.4, 3.1.3, 3.1.6, 3.2.5
diagonal of a ~	2.3.6, 3.1.4, 3.2.5, 5.7.2, 5.8.2, 6.7.2, 6.8.2
row of a ~	1.1.4, 3.1.3, 3.1.6, 3.2.5
transpose of a ~	2.3.6, 3.2.5, 5.7.1, 5.8.2, 6.7.1, 6.8.2
zero~	1.1.2, see *zeromat*
message	2.1.6, 4.2, 6.0.9
mode	2.1.2, 2.1.3, 2.2, 2.2.7
~ declaration	see declaration
~ equivalencing	2.2.4
mode/operator package	2.2, 2.2.5, 2.2.6, 2.2.7
modop	2.2, 2.2.5, 2.2.6, 2.2.7
module (mathem.)	1.1.2, 1.1.3, 3.1.2
module (progr. language)	2.2.6
monadic operation	see operation
monoid	1.1.1
multiple value	see value
multiplication	1.1.1, 4.3.3
array generating ~	3.2.10, 5.16, 5.18, 6.16, 6.18
level1 assigning ~	3.2.7, 5.13, 6.13
matrix column ~	3.2.10, 5.18.1, 6.18.1
matrix matrix ~	3.2.10, 5.18, 6.18
matrix vector ~	3.2.10, 5.18.1, 6.18.1
row matrix ~	3.2.10, 5.18.1, 6.18.1
name	see reference
natural number (N)	1.1, 1.1.1, 2.2.3
norm	3.2.3, 3.2.9
euclidean vector ~	3.2.9
operation	
array assigning ~	3.2.7, 1.3.2
array generating ~	3.2.2, 3.2.7, 5.2, 6.2, 1.3.2
assigning ~	1.3.2
binary ~	see dyadic operation
concrete ~	1.3.1
~ declaration	see declaration
dyadic ~	1.1
elementwise ~	2.3.7, 3.2.7
hidden ~	see hidden
level1 assigning ~	see level1
level2 assigning ~	see level2
~ library	see mode/operator package
monadic ~	1.1

operator	
exchange ~	see exchange
identity ~	see identity
linear ~	1.1.2
optimization	2.1.5, 2.3.7, 6.0
overflow	2.2.3
int~	2.2.3
package	see mode/operator package
polynomial	2.2.7
cauchy power of ~	3.2.10, 5.18.4, 6.18.4
cauchy product of ~	3.2.10, 5.18.4, 6.18.4
composition of ~	3.2.10, 5.18.5, 6.18.5
convolution product of ~	3.2.10, 5.18.4, 6.18.4
derivative of a ~	3.2.10, 5.18, 6.18
value of ~	3.2.9, 5.17.4, 6.17.4
pragmat	2.1.4, 2.1.5
precision	2.2.2, see *real*, *compl*
prelude	see mode/operator package
library ~	2.1.2, 3.1
standard ~	2.1.2, 3.1.1
procedure	
array generating ~	2.3.3, 5.1, 6.1
~ declaration	see declaration
generic ~	2.2
projection	1.3.1, 2.3.5, 3.2.5
rational function	see polynomial
rational number ($\mathbb{Q}$)	1.1, 1.1.1, 1.2.1, 2.2.3, 4.3.3
rational (*rat*)	2.2.3, 3.1.2, 4.3.3
real number ($\mathbb{R}$)	1.1, 1.1.1, 1.2.1, 2.2.2, 4.3.3
real	2.2.1, 2.2.2, 3.1.2, 4.3.3
ref ref (higher level of reference)	2.3, 3.1.1, 3.1.4
reference (name, address)	2.3, 2.3.4, 3.1.4, 3.1.5
depth ~	2.3, 3.1.4, 3.3.2
reverse sumproduct	see sumproduct
ring	1.1.1, 2.2.3, 3.1.2
row	3.1.4, 3.2.5, 3.2.10, 5.7.4, 5.8.4, 6.7.4, 6.8.4
~ of a matrix	see matrix
~ display	see display

scalar	1.1.2, 3.1.2
~ field	see field
~ system	1.1.1, 1.2.1, 2.1.1, all of 2.2
scope	2.1.3, 3.1.5
selection	2.3.5, 3.1.3
selector	2.3.5, 3.1.6, 3.2.5, 5.0, 6.0
destination ~	2.3.5, 3.3.3, 5.0.8, 6.0.8
source ~	2.3.5, 3.1.6, 5.0.7, 6.0.7
semigroup	1.1.1
slice (slicer, slicing)	2.3, 2.3.5, 3.1.3, 3.1.5, 3.1.6, 3.2.5
source	3.1.5
space	
euclidean ~	1.1.5
innerproduct ~	1.1.5
vector ~	1.1.2, 2.1.1, 3.1.2
unitary ~	1.1.5
subscript(ing)	2.3.5, 2.3.6, 3.1.3
substraction	4.3.3, see addition
sum (summation)	1.1.2, 3.2.6, 5.10, 6.10
sumproduct	3.2.9, 5.17, 6.17
systemparameter	2.2.5
taboo (mark, token)	2.1.4, 2.2, see hidden
TORRIX	1.2, 2.1.2
TORRIX68	2.1.2, 3.1
TORRIX68S	2.3.3
TORRIX-BASIS	2.1.4, all of 5, all of 6
TORRIX-BASIS LEVEL1	2.3.2, 2.3.3, all of 3.2
TORRIX-REAL	2.2.1
TORRIX message system	see message
TORRIX postlude	4.2
torrix	3.1.4, 3.2.5
total	
~ array	see array
~ extrema	see extrema
~ selector	see selector
transpose of a matrix	see matrix
transput	2.1.6

trimmer 2.3.5, 3.1.3, see *trimmer*

trimming operation 3.3.4, 5.9, 6.9

upperbound 2.1.5, 3.1.3

value 3.1.4, 3.1.5, 3.2.5
~ interrogation 3.2.3, 5.4, 6.4
multiple ~ 2.3, 3.1.1, 3.1.3, 5.0.1
new ~ 3.2.5, 5.5, 6.5

variability 2.3.1, see level1, level2

variable 3.1.5
level1 ~ see level1
level2 ~ see level2
subscripted ~ 3.1.5

vector 1.1.2, 1.2.4, 2.1.1, see *vec*
column ~ see column
row ~ see row
~ space see space
zero ~ 1.1.2, see *zerovec*

virtual
~ bounds see bounds
~ part 3.1.4, 4.1
~ zeros 3.1.4, 4.1

weak dereferencing see dereferencing

widening 2.2.1, 2.2.3, 3.1.5, 3.2.5, see *widen*

zero
~ matrix see matrix
~ transformation 1.1.4
~ vector see vector

TORRIX modes

array1	3.1.3	3.1.4		5.0.1
array2	3.1.3	3.1.4		5.0.1
coarray1	3.1.3	3.1.4		volume II
coarray2	3.1.3	3.1.4		volume II
comat	3.1.1	3.1.4		volume II
coscal	3.1.2		4.3.4	volume II
covec	3.1.1	3.1.4		volume II
index	3.1.3			5.0.2
int	3.1.2			
intarray	3.1.3	3.1.4		5.0.1
mat	3.1.1	3.1.4		5.0.2
pair	3.1.6	3.3.3		5.0.3
scal	3.1.2		4.3.3	
trimmer	3.1.6	3.3.3		5.0.3
vec	3.1.1	3.1.4		5.0.2

TORRIX operators

abs	7.1.2.10	7.2.2.10				
col	3.2.5	5.7.3	6.7.3			
compat	3.2.3	5.4.4	6.4.4			
conj	4.3.4	volume II				
copy	3.2.2	5.2.1	6.2.1			
copycol	3.2.5	5.8.3	6.8.3			
copydiag	3.2.5	5.8.2	6.8.2			
copyrow	3.2.5	5.8.4	6.8.4			
copytrnsp	3.2.5	5.8.1	6.8.1			
deriv	3.2.10	5.18.6	5.18.7	6.18.6	6.18.7	
diag	3.2.5	5.7.2	6.7.2			
equ	3.2.3	5.4.3	6.4.3			
fitsin	3.2.3	5.0.6	5.3.7	6.0.6	6.3.7	
frac	7.1.2.9	7.2.2.9				
gcd	7.1.3.1	7.2.3.1				
identy	3.2.5	5.5.5	6.5.5			
im	4.3.4	volume II				
inspan	3.2.2	5.2.4	6.2.4			
into	3.2.5	5.5.1	5.5.2	5.5.3	5.5.4	
		6.5.1	6.5.2	6.5.3	6.5.4	
lwb	3.2.3	5.3.1	5.3.4	6.3.1	6.3.4	
max	3.1.1	3.2.6	4.3.1	5.10.3	5.11.1	5.11.5
				6.10.3	6.11.1	6.11.5
maxabs	3.2.6	5.10.5	5.11.3	6.10.5	6.11.3	
meet	3.2.2	5.2.3	6.2.3			
min	3.1.1	3.2.6	4.3.1	5.10.4	5.11.2	5.11.6
				6.10.4	6.11.2	6.11.6
minabs	3.2.6	5.10.6	5.11.4	6.10.6	6.11.4	
multrnsp	3.2.10	5.18.3	6.18.3			

neg	3.2.7	5.13.4	6.13.4			
o	3.2.9	3.2.10	5.17.4	5.18.5	6.17.4	6.18.5
re	4.3.4	volume II				
row	3.2.5	5.7.4	6.7.4			
search	3.2.3	5.4.5	6.4.5			
sigma	3.2.6	5.10.1	6.10.1			
sigmabs	3.2.6	5.10.2	6.10.2			
size	3.2.3	5.3.3	5.3.6	6.3.3	6.3.6	
span	3.2.2	5.2.2	6.2.2			
square	3.2.3	5.3.8	6.3.8			
subscr	3.2.2	3.2.5	5.2.5	5.2.6	6.2.5	6.2.6
trim	3.3.4	5.9.2	6.9.2			
trims	3.3.4	5.9.1	6.9.1			
trnsp	3.2.5	5.7.1	6.7.1			
trnspmul	3.2.10	5.18.2	6.18.2			
upb	3.2.3	5.3.2	5.3.5	6.3.2	6.3.5	
widen	3.1.2	4.3.3	4.3.4			
zero	3.2.3	5.4.1	6.4.1			

+	3.2.8	5.14.1	6.14.1				
−	3.2.8	5.14.2	5.14.3	6.14.2	6.14.3		
×	3.2.9	3.2.10	5.16.1	5.17.1	5.18.1		
			6.16.1	6.17.1	6.18.1		
/	3.2.10	5.16.2	6.16.2				
<>	3.2.9	5.17.2	6.17.2				
><	3.2.9	5.17.3	6.17.3				
××	3.2.10	5.18.4	6.18.4				
+<	3.2.7	5.12.1	5.12.3	6.12.1	6.12.3		
−<	3.2.7	5.12.2	5.12.4	6.12.2	6.12.4		
×<	3.2.7	5.13.1	6.13.1				
/<	3.2.7	5.13.3	6.13.3				
×>	3.2.7	5.13.2	5.13.5	6.13.2	6.13.5		
/>	3.2.7	5.13.6	6.13.6				
+:=	3.3.5	5.15.1	6.15.1				
−:=	3.3.5	5.15.2	6.15.2				
=:=	3.1.1	3.2.5	4.3.5	5.6.1	5.6.2	6.6.1	6.6.2
=	3.2.3	5.4.2	6.4.2	7.1.5.1	7.2.5.1		
/=	3.2.3	5.4.2	6.4.2	7.1.5.2	7.2.5.2		
?	3.1.6	5.0.4	5.0.7	6.0.4	6.0.7		
!	3.3.3	5.0.4	5.0.8	6.0.4	6.0.8		
//	3.1.6	3.3.3	5.0.5	6.0.5			

TORRIX identifiers

copyerrorfile	proc void	4.2.3		
errorfile	ref file	4.2.1		
errorfile is open	ref bool	4.2.1		
fatal	bool	4.2.2		
genallowance	proc(bool)void	3.2.1	4.3.2	
genarray1	proc(int,int)vec	3.2.2	5.1.2	6.1.2
genarray2	proc(int,int,int,int)mat	3.2.2	5.1.3	6.1.3
genindex	proc(int)index	3.2.2	5.1.4	6.1.4
genintarray	proc(int,int)index	3.2.2	5.1.1	6.1.1
genmat	proc(int,int)mat	3.2.2	5.1.6	6.1.6
gensquare	proc(int)mat	3.2.2	5.1.7	6.1.7
genvec	proc(int)vec	3.2.2	5.1.5	6.1.5
length errorfile	int	4.2.1		
maxdex	int	3.2.1	4.3.2	
mindex	int	3.2.1	4.3.2	
number of warnings	proc int	4.2.2		
reset number of warnings	proc void	4.2.2		
setgendex	proc(int,int)void	3.2.1	4.3.2	
stop	label	4.2.6		
stringindexbounds	proc(index)[]char	4.2.5		
stringmatbounds	proc(mat)[]char	4.2.5		
stringparam2	proc(int,int)[]char	4.2.5		
stringparam4	proc(int,int,int,int)[]char	4.2.5		
stringvecbounds	proc(vec)[]char	4.2.5		
torrix	proc(bool,[]char)void	4.2.4		
warning	bool	4.2.2		
zeromat	mat	3.2.2	5.0.2	
zerovec	vec	3.2.2	5.0.2	
†*maxgendex*	ref int	3.2.1	4.3.2	
†*mingendex*	ref int	3.2.1	4.3.2	
†*numberwarnings*	ref int	4.2.2		
†*text1* --- †*text23*	[]char	6.0.9		

BIBLIOGRAPHY

{01} BARKER, V.A. (ed.), *Sparse matrix techniques*, Lecture Notes in Mathematics, vol. 572, Springer Verlag, New York, Heidelberg, Berlin, 1977.

{02} BUNCH, J.R. & D.J. ROSE (eds.), *Sparse matrix computations*, Academic Press, New York, 1976.

{03} CONTROL DATA, *ALGOL68, Version I, Reference manual*, CONTROL DATA B.V., Rijswijk, The Netherlands, 1975.

{04} DAHL, O.J., E.W. DIJKSTRA & C.A.R. HOARE, *Structured programming*, Academic Press, New York, 1972.

{05} DEKKER, T.J. & W. HOFFMANN, *ALGOL60 procedures in numerical algebra, parts 1 & 2*, Mathematical Centre Tracts 22 & 23, Amsterdam, 1968.

{06} DIJKSTRA, E.W., *A discipline of programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

{07} GERBIER, A., *Mes premières constructions de programmes*, Lecture Notes in Computer Science, vol. 55, Springer Verlag, New York, Heidelberg, Berlin, 1977, (in French).

{08} GUTTAG, J.V., *Abstract data types and the development of datastructures*, Comm. ACM 20 (1977) 396-404.

{09} HALMOS, P.R., *Finite dimensional vector spaces*, 2nd edition, P. van Nostrand Company, 1958.

{10} IVERSON, K.E., *A programming language*, John Wiley & Sons, New York, 1962.

{11} KNUTH, D.E., *The art of computer programming*, vol. II: *Seminumerical algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

{12} KOSTER, C.H.A., *Visibility and types*, Proc. Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, special issue, (1976) 179-190.

{13} LEARNER, A. & A.J. POWELL, *An introduction to ALGOL68 through problems*, Macmillan Computer Science series, Macmillan Press, London, 1974.

{14} LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal Introduction to ALGOL68*, revised edition, North Holland Publ. Company, Amsterdam, 1977.

{15} LISKOV, B. & S. ZILLES, *Programming with abstract data types*, Proc. Symp. on very high level languages, SIGPLAN Notices 9 (April 1974) 50-59.

{16} MAC LANE, S., *Categories for the working mathematician*, Springer Verlag, New York, Heidelberg, Berlin, 1971.

{17} MAC LANE, S. & G. BIRKHOFF, *Algebra*, Macmillan Company, New York, 1967.

{18} MEERTENS, L.G.L.T., *Abstracte datatypen*, in {33} (in Dutch).

{19} MEERTENS, L.G.L.T., *Procedurele datastructuren*, in {33} (in Dutch).

{20} MEULEN, S.G. VAN DER, *TORRIX (vecTOR-matRIX), on the Manipulation of (possibly sparse) Vectors and Matrices over Arbitrary Fields*, in: RAYWARD-SMITH, V.J. (ed.), Proc. Conf. on Applications of ALGOL68, 1976, 80-90.

{21} MEULEN, S.G. VAN DER, *ALGOL68 Might-have-beens*, Proc. Strathclyde ALGOL68 Conf., SIGPLAN Notices 12 (June 1977) 1-18.

{22} MEULEN, S.G. VAN DER, *Refers, a generalization of address and access algorithms*, (in preparation).

{23} MEULEN, S.G. VAN DER & P. KÜHLING, *Programmieren in ALGOL68*, vol. I & II, Walter de Gruyter, Berlin, 1974, 1976, (in German).

{24} MEULEN, S.G. VAN DER & M. VELDHORST, *Datastructuren voor lineaire ruimten, "TORRIX"*, in {33} (in Dutch).

{25} PAGAN, F.G., *A practical guide to ALGOL68*, John Wiley & Sons, New York, 1976.

{26} PAUL, G. & M.W. WILSON, *The vectran language*, IBM Palo Alto Scientific Center, Palo Alto, California, 1975.

{27} SCHUMAN, S.A. (ed.), Proc. Int. Symp. on Extensible Languages, SIGPLAN Notices 6 (December 1971) / IBM-France Centre Scientifique de Grenoble, Report no. FF2.0143 (1971).

{28} SCHUMAN, S.A., *Toward modular programming in high-level languages*, ALGOL Bulletin 37.4.1 (1974) 30-53.

{29} SCHUMAN, S.A. (ed.), *New directions in algorithmic languages 1975*, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975.

{30} SCHUMAN, S.A. (ed.), *New directions in algorithmic languages 1976*, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, 1976.

{31} SHAW, M. & W.A. WULF, *Abstraction and verification in ALPHARD: defining and specifying iteration and generators*, Comm. ACM 20 (1977) 553-563.

{32} TANENBAUM, A.S., *A tutorial on ALGOL68*, ACM Computing Surveys 8 (June 1976) 155-180.

{33} VLIET, J.C. VAN, *Colloquium Capita Datastructuren*, Mathematical Centre Syllabus 37, Amsterdam, 1978 (in Dutch).

{34} WILKINSON, J.H., *The algebraic eigenvalue problem*, Oxford University Press, London, 1965.

{35} WILKINSON, J.H. & C. REINSCH, *Handbook for automatic computation*, vol. II, *Linear Algebra*, Springer Verlag, New York, Heidelberg, Berlin, 1971.

{36} WIJNGAARDEN, A. VAN et al., *Revised report on the algorithmic language ALGOL68*, Mathematical Centre Tract 50, Amsterdam, 1976,
Springer Verlag, New York, Heidelberg, Berlin, 1976.
SIGPLAN Notices, ACM, Vol. 12, Numb. 5, May 1977.

{37} SIGPLAN NOTICES, ACM, Vol. 12, Numb. 5, May 1977
.1 *Revised Report on the algorithmic language ALGOL68*
A. van Wijngaarden et al.
.2 *A sublanguage of ALGOL68*
P.G. Hibbard
.3 *The report on the standard hardware representation for AGLOL68*
W.J. Hansen & H. Boom

OTHER TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

A leaflet containing an order-form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam-1005, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distributions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.TH.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54 and 67.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAERL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Excercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium*, 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws*, 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.

* MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1.* ISBN 90 6196 109 2.

* MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2.* ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.

* MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2.* ISBN 90 6196 119 X.

MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.

MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977. ISBN 90 6196 125 4.

MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: semantics and proof theory*, 1976. ISBN 90 6196 127 0.

MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976. ISBN 90 6196 129 7.

MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977. ISBN 90 6196 130 0.

* MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic.* ISBN 90 6196 122 x.

MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.

MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.

MCT 76 M. REM, *Associons and the closure statement*, 1976. ISBN 90 6196 135 1.

MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977 ISBN 90 6196 134 3.

MCT 78 E. DE JONGE, A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977. ISBN 90 6196 133 5.

MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rankstatistics*, 1977. ISBN 90 6196 145 9.

MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.

MCT 81 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part I, 1976. ISBN 90 6196 140 8.

MCT 82 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part II, 1976. ISBN 90 6196 141 6.

* MCT 83 L.S. VAN BENTEM JUTTING, *Checking Landau's "Grundlagen" in the automath system*, ISBN 90 6196 147 5.

MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977. ISBN 90 6196 148 3.

MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977. ISBN 90 6196 151 3.

MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I*, 1978. ISBN 90 6196 152 1.

* MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*, ISBN 90 6196 153 x.

MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977. ISBN 90 6196 154 8.

MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978.
ISBN 90 6196 155 6.

* MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, . ISBN 90 6196 156 4.

* MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, . ISBN 90 6196 157 2.

* MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, .
ISBN 90 6196 158 0.

MCT 93 H.C. TIJMS (ed.) & J. WESSELS (ed.), *Markov decision theory*, 1977.
ISBN 90 6196 160 2.

* MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, . ISBN 90 6196 162 9.

* MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, .
ISBN 90 6196 163 7.

* MCT 96 P.M.B. VITÁNYI, *Lindenmayer systems: structure, languages, and growth functions*, . ISBN 90 6196 164 5.

* MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.

* MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, .
ISBN 90 6196 166 1

* MCT 99 J.K. Lenstra, A.H.G. Rinnoy Kan & P. Van Emde Boas, *Interfaces between computer science and operations research*, .
ISBN 90 6196 170 X.

* MCT 100 P.C. Baayen, D. van Dulst, & J. Oosterhoff, *Proceedings bicentennial congress of the Wiskundig Genootschap , part 1*, .
ISBN 90 6196 168 8.

* MCT 101 P.C. Baayen, D. van Dulst, & J. Oosterhoff, *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, .
ISBN 90 9196 169 6.

AN ASTERISK BEFORE THE NUMBER MEANS "TO APPEAR"