Institut für
Angewandte
Mathematik

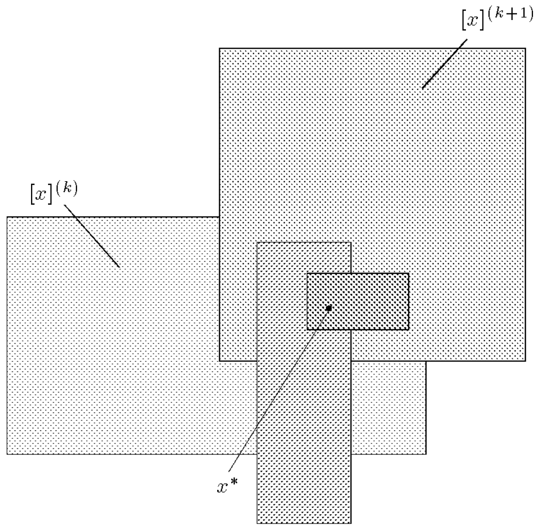
Universität Karlsruhe (TH)
D-76128 Karlsruhe

A Survey of PASCAL–XSC and a Language Reference Supplement on Dynamic and Flexible Arrays

Peter Januschke, Dietmar Ratz

F orschungsschwerpunkt
C omputerarithmetik,
I ntervallrechnung und
N umerische Algorithmen mit
E rgebnisverifikation

Bericht 1/1998



Impressum

Herausgeber:	Institut für Angewandte Mathematik Lehrstuhl Prof. Dr. Ulrich Kulisch Universität Karlsruhe (TH) D-76128 Karlsruhe
Redaktion:	Dr. Dietmar Ratz

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über

`ftp://iamk4515.mathematik.uni-karlsruhe.de`

im Verzeichnis: `/pub/documents/reports`

oder über die World Wide Web Seiten des Instituts

`http://www.uni-karlsruhe.de/~iam`

Autoren-Kontaktadresse

Rückfragen zum Inhalt dieses Berichts bitte an

Dietmar Ratz
Institut für Angewandte Mathematik
Universität Karlsruhe (TH)
D-76128 Karlsruhe

E-Mail: Dietmar.Ratz@math.uni-karlsruhe.de

A Survey of PASCAL–XSC and a Language Reference Supplement on Dynamic and Flexible Arrays

Peter Januschke, Dietmar Ratz

Contents

Zusammenfassung

Ein PASCAL–XSC–Überblick und eine Sprachbeschreibungs-Ergänzung: PASCAL–XSC ist eine universelle Programmiersprache, die außerdem speziell die Implementierung von hochentwickelten numerischen Algorithmen unterstützt. Das PASCAL–XSC System hat den Vorteil der Portabilität auf verschiedenen Plattformen (Personal Computer, Workstations, Großrechner und Supercomputer) durch einen portablen Compiler, der nach ANSI-C übersetzt.

Mittels der mathematischen Module von PASCAL–XSC können numerische Algorithmen, die hochgenaue und automatisch verifizierte Ergebnisse liefern, sehr leicht programmiert werden. PASCAL–XSC vereinfacht das Design von Programmmen in den Ingenieurwissenschaften und im wissenschaftlichen Rechnen durch modulare Programmstruktur, benutzerdefinierte Operatoren, Überladen von Funktionen, Prozeduren und Operatoren, Funktionen und Operatoren mit allgemeinem Ergebnistyp und dynamische Felder. Arithmetische Standard Module für zusätzliche numerische Datentypen (inclusive Operatoren und Standardfunktions von hoher Genauigkeit) und die exakte Ausdrucksauswertungn stellen die wichtigsten numerischen Tools dar.

In PASCAL–XSC geschriebene Programme sind leicht lesbar, da alle Operationen, auch die in höheren mathematischen Räumen, als Operatoren realisiert sind und in der üblichen mathematischen Notation verwendet werden können.

In aktuellen Compiler-Versionen von PASCAL–XSC wurde das Konzept der dynamischen Felder beträchtlich erweitert. Ein Benutzer kann nun dynamische Felder mehrfach und mit unterschiedlicher Größe zur Laufzeit seines Programmes allokieren. Darüberhinaus können dynamische Felder auch als Komponenten anderer PASCAL Strukturen wie Records und statische Felder vereinbart werden.

Abstract

A Survey of PASCAL–XSC and a Language Reference Supplement: PASCAL–XSC is a general purpose programming language which provides special support for the implementation of sophisticated numerical algorithms. The new PASCAL–XSC system has the advantage of being portable across many platforms and is available for personal computers, workstations, mainframes and supercomputers by means of a portable compiler which translates to ANSI-C language.

By using the mathematical modules of PASCAL–XSC, numerical algorithms which deliver highly accurate and automatically verified results can be programmed easily. PASCAL–XSC simplifies the design of programs in engineering and scientific computation by modular program structure, user-defined operators, overloading of functions, procedures, and operators, functions and operators with arbitrary result type and dynamic arrays. Arithmetic standard modules for additional numerical data types including operators and standard functions of high accuracy and the exact evaluation of expressions provide the main numerical tools.

Programs written in PASCAL–XSC are easily readable since all operations, even those in the higher mathematical spaces, have been realized as operators and can be used in conventional mathematical notation.

In current compiler versions of PASCAL–XSC, the concept of dynamic arrays has been significantly extended. A user is now able to allocate a dynamic array variable several times and with different size during the execution of his or her program. Moreover, dynamic arrays may now be declared as components of other PASCAL structures such as records or static arrays.

1 Introduction

In the last decades continuous efforts have been made to enhance the power of programming languages. New powerful languages have been designed, and the enhancement of existing languages such as Fortran is in constant progress. However, since many of these languages still lack a precise definition of their arithmetic, the same program may produce different results on different processors.

These days, the elementary arithmetic operations of electronic computers are usually floating-point operations of highest accuracy. In particular, this means that for any choice of operands, the computed result is the rounded exact result of the operation with just one final rounding applied. See the IEEE Arithmetic Standard [4] as an example. This arithmetic standard also requires the four basic arithmetic operations $+, -, *$, and / with directed roundings. A large number of processors already provide these operations, but only few programming languages allows easy access to them.

On the other hand, there has been a noticeable shift in scientific computation from general purpose computers to vector and parallel computers. These so-called supercomputers provide additional arithmetic operations such as "multiply and add", "accumulate" or "multiply and accumulate" (see [11]). These hardware operations should always deliver a result of highest accuracy, but as of yet, no processor which fulfills this requirement is available. In some cases, the results of numerical algorithms computed on vector computers are totally different from the results computed on the same processor in scalar mode (see [15],[30]).

PASCAL–XSC is the result of a long-term venture by a team of scientists to produce a powerful tool for solving scientific problems. The mathematical definition of the arithmetic is an intrinsic part of the language, including optimal arithmetic operations with directed roundings which are directly accessible in the language. Further arithmetic operations for intervals and complex numbers and even vector/matrix operations provided by precompiled arithmetic modules are defined with maximum accuracy according to the rules of semimorphism (see [25]).

The development of PASCAL–XSC programs is supported by the PASCAL–XSC *development system* [3] consisting of the PASCAL–XSC *compiler* [2] and the PASCAL–XSC *runtime system* [12] which are both written in ANSI C [5]. Instead of implementing a large variety of "native code generators" for different processor and operating systems, the PASCAL–XSC system compiles a given PASCAL–XSC source code into C code which is passed to a C compiler. Finally, the resulting object code and the routines of the PASCAL–XSC runtime system are linked together. Because of the wide distribution of C compilers, the PASCAL–XSC system is *available* on many computers (see section 3.3). Both the PASCAL–XSC source code and the generated C code are *portable*.

From the point of view of mathematics, it is of fundamental importance that results of implemented algorithms are reproducible in spite of different computing facilities. Unfortunately, the arithmetical capabilities of computer systems are quite different concerning the representation of floating-point numbers and the way arithmetic operations are processed. Therefore, a common accurate arithmetical basis must be supported by a programming language. The PASCAL–XSC runtime system comprises a complete set of routines which is based on the IEEE 754 binary floating-point arithmetic standard [4]. All arithmetic operations are implemented in software and do not depend on the actual operations of the processor in use nor on the C runtime system. To achieve better performance, the runtime system can be configured in such a way that it adapts to the arithmetic hardware unit of the processor in use.

2 The Language PASCAL–XSC

PASCAL–XSC is an eXtension of the programming language PASCAL for Scientific Computation, containing the following features:

- Standard PASCAL
- Universal operator concept (user-defined operators)
- Functions and Operators with arbitrary result type
- Overloading of procedures, functions and operators
- Overloading of assignment operator
- Overloading of the I/O -routines *read* and *write*
- Module concept
- Dynamic arrays
- Access to subarrays
- String concept
- Controlled rounding
- Optimal (exact) scalar product
- Standard type *dotprecision* (a fixed-point format covering the whole range of floating-point products)
- Additional arithmetic standard types such as *complex, interval,* etc.
- Highly accurate arithmetic for all standard types
- Highly accurate standard functions
- Exact evaluation of expressions (#-expressions)

A complete description of the language PASCAL–XSC and the arithmetic modules as well as a collection of sample programs is given in [21] and [22]. A short survey of the language features is given in the following sections. Moreover, the extended concept of dynamic and flexible arrays contained in current versions (Version 3.0 and higher) of PASCAL–XSC is described in detail in Appendix B.

2.1 Standard Data Types, Predefined Operators, and Functions

In addition to the integer and real data types of standard PASCAL, the following numerical data types are available in PASCAL–XSC:

	complex	*interval*	*cinterval*
rvector	*cvector*	*ivector*	*civector*
rmatrix	*cmatrix*	*imatrix*	*cimatrix*

where the prefix letters *r*, *i*, and *c* are abbreviations for *real*, *interval*, and *complex*. So *cinterval* means *complex interval* and, for example, *cimatrix* denotes complex interval matrices, whereas *rvector* specifies real vectors. The vector and matrix types are defined as dynamic arrays and can be used with arbitrary index ranges.

A large number of operators are predefined for these types in the arithmetic modules of PASCAL–XSC (see section 2.9). All of these operators deliver results with maximum accuracy.

Compared to standard PASCAL, there are 11 new operator symbols. These are the operators $\circ<$ and $\circ>$, $\circ \in \{+,-,*,/\}$ for operations with downwardly and upwardly directed rounding and the operators $**, +*, ><$ needed in interval computations for the intersection, the convex hull, and the disconnectivity test.

right operand / left operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
monadic[1]	$+, -$	$+, -$	$+, -$	$+, -$	$+, -$	$+, -$
integer real complex	$+, +<, +>, -, -<, ->, *, *<, *>, /, /<, />, +*$	$+, -, *, /, +*$	$*, *<, *>$	$*$	$*, *<, *>$	$*$
interval cinterval	$+, -, *, /, +*$	$+, -, *, /, +*, **$	$*$	$*$	$*$	$*$
rvector cvector	$*, *<, *>, /, /<, />$	$*, /$	$+, +<, +>,$[2] $-, -<, ->, *, *<, *>, +*$	$+, -, *,$[2] $+*$		
ivector civector	$*, /$	$*, /$	$+, -, *,$[2] $+*$	$+, -, *,$[2] $+*, **$		
rmatrix cmatrix	$*, *<, *>, /, /<, />$	$*, /$	$*, *<, *>$	$*$	$+, +<, +>,$[2] $-, -<, ->, *, *<, *>, +*$	$+, -, *,$[2] $+*$
imatrix cimatrix	$*, /$	$*, /$	$*$	$*$	$+, -, *,$[2] $+*$	$+, -, *,$[2] $+*, **$

[1]) The operators of this row are monadic (i.e. there is no left operand).

[2]) $*$ denotes the scalar or matrix product.

$+*$: Interval hull

$**$: Interval intersection

Table 1: Predefined Arithmetic Operators

Tables 1 and 2 show all predefined arithmetic and relational operators in connection with the possible combinations of operand types.

right operand left operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
integer real complex	=, <>, <=, <, >=, >	**in** =, <>				
interval cinterval	=, <>	**in**, ><,[1)] =, <>, <=, <, >=, >				
rvector cvector			=, <>, <=, <, >=, >	**in** =, <>		
ivector civector			=, <>	**in**, ><,[1)] =, <>, <=, <, >=, >		
rmatrix cmatrix					=, <>, <=, <, >=, >	**in** =, <>
imatrix cimatrix					=, <>	**in**, ><,[1)] =, <>, <=, <, >=, >

[1)] The operators <= and < denote the "subset" relations, >= and > denote the "superset" relations.

>< : Test of disconnectivity for intervals

in : Test of membership of a point in an interval or test on strict enclosure of an interval in the interior of an interval

Table 2: Predefined Relational Operators

Compared with standard PASCAL, PASCAL–XSC provides an extended set of mathematical standard functions (see table 3). These functions are available for the types *real*, *complex*, *interval*, and *cinterval* with a generic name and deliver a result of maximum accuracy. The functions for the types *complex*, *interval*, and *cinterval* are provided in the arithmetic modules of PASCAL–XSC.

	Function	Generic Name
1	Absolute Value	abs
2	Arc Cosine	arccos
3	Arc Cotangent	arccot
4	Inverse Hyperbolic Cosine	arcosh
5	Inverse Hyperbolic Cotangent	arcoth
6	Arc Sine	arcsin
7	Arc Tangent	arctan
8	Inverse Hyperbolic Sine	arsinh
9	Inverse Hyperbolic Tangent	artanh
10	Cosine	cos
11	Cotangent	cot
12	Hyperbolic Cosine	cosh
13	Hyperbolic Cotangent	coth
14	Exponential Function	exp
15	Power Function (Base 2)	exp2
16	Power Function (Base 10)	exp10
17	Natural Logarithm (Base e)	ln
18	Logarithm (Base 2)	log2
19	Logarithm (Base 10)	log10
20	Sine	sin
21	Hyperbolic Sine	sinh
22	Square	sqr
23	Square Root	sqrt
24	Tangent	tan
25	Hyperbolic Tangent	tanh

Table 3: Predefined Mathematical Functions for the types *integer, real, complex, interval,* and *cinterval*

Besides the mathematical standard functions, PASCAL–XSC provides the necessary type transfer functions *intval, inf, sup, compl, re*, and *im* for conversion between the numerical data types (for scalar and array types).

2.2 The General Operator Concept

By a simple example of interval addition, the advantages of a general operator concept are demonstrated. In the absence of user-defined operators, there are two ways to implement the addition of two variables of type *interval* declared by

```
type interval = record inf, sup: real; end;
```

One can use a procedure declaration (operators with directed rounding such as $+<$ and $+>$ are not available in standard PASCAL)

```
procedure intadd (a, b: interval; var c: interval);
begin
 c.inf := a.inf +< b.inf;
 c.sup := a.sup +> b.sup
end;
```

mathematical notation	corresponding program statements

$z := a + b + c + d$	intadd(a,b,z);
	intadd(z,c,z);
	intadd(z,d,z);

or a function declaration (only possible in PASCAL–XSC, not in standard PASCAL)

```
function intadd (a, b: interval): interval;
begin
  intadd.inf := a.inf +< b.inf;
  intadd.sup := a.sup +> b.sup
end;
```

mathematical notation	corresponding program statement
$z := a + b + c + d$	z := intadd(intadd(intadd(a,b),c),d);

In both cases the transcription of the mathematical formula looks rather complicated. By comparison, if one implements an operator in PASCAL–XSC ,

```
operator + (a, b: interval) intadd: interval;
begin
 intadd.inf := a.inf +< b.inf;
 intadd.sup := a.sup +> b.sup
end;
```

mathematical notation	corresponding program statement
$z := a + b + c + d$	z := a + b + c + d;

then multiple addition of intervals is described in the traditional mathematical notation. Besides the possibility of overloading operator symbols, one is allowed to use named operators. The declaration of such operators must be preceded by a priority declaration. There exist four different levels of priority, each represented by its own symbol:

• monadic	:	$\uparrow$	level 3 (highest priority)
• multiplicative	:	$*$	level 2
• additive	:	$+$	level 1
• relational	:	$=$	level 0

For example, an operator for the calculation of the binomial coefficient $\binom{n}{k}$ may be defined in the following manner:

```
priority choose = *;  { priority declaration }

operator choose (n, k: integer) binomial: integer;
var i, r : integer;
begin
  if k > n div 2 then k := n - k;
  r := 1;
  for i := 1 to k do
    r := r * (n - i + 1) div i;
  binomial := r;
end;
```

mathematical notation	corresponding program statement
$c := \binom{n}{k}$	c := n choose k

The operator concept realized in PASCAL–XSC offers the possibilities of

- defining an arbitrary number of operators
- overloading operator symbols or operator names arbitrarily many times
- implementing recursively defined operators

Also, PASCAL–XSC offers the possibility of overloading the assignment operator := to allow a natural notation for assignments:

Example:

```
var
  c : complex;
  r : real;

...

operator := (var c: complex; r: real);
begin
  c.re := r;
  c.im := 0;
end;

...

r := 1.5;
c := r;    { complex number with real part 1.5 and imaginary part 0 }
```

2.3 Overloading of Subroutines

Standard PASCAL provides the mathematical standard functions

sin, *cos*, *arctan*, *exp*, *ln*, *sqr*, and *sqrt* for numbers of type *real* only. In order to implement the sine function for interval arguments, a new function name like *isin(...)* must be used because overloading of the standard function name *sin* is not allowed in standard PASCAL.

In contrast, PASCAL–XSC allows overloading of function and procedure names, whereby a generic symbol concept is introduced into the language. So the symbols

sin, cos, arctan, exp, ln, sqr, and *sqrt*

can be used not only for arguments of type *real*, but also for intervals, complex numbers, and other types. To distinguish between overloaded functions or procedures with the same name, the number and type of their arguments is used, similar to the method for operators. The type of the result, however, is *not* used.

Example:

```
procedure rotate (var a, b: real);
procedure rotate (var a, b, c: complex);
procedure rotate (var a, b, c: interval);
```

The overloading concept also applies to the standard procedures *read* and *write* in a slightly modified way. The first parameter of a newly declared input/output procedure must be a **var**-parameter of a file type and the second parameter represents the quantity that is to be input or output. All further parameters are interpreted as format specifications.

Example:

```
procedure write (var f: text; c: complex; w:  integer);
begin
  write (f, '(', c.re : w, ',', c.im :  w, ')');
end;
```

When calling an overloaded input/output procedure, the file parameter may be omitted which corresponds to a call with one of the standard files *input* or *output*. The format parameters must be introduced and separated by colons. Moreover, several input or output statements can be combined into a single statement just as in standard PASCAL.

Example:

```
var
  r : real;
  c : complex;

...

write (r : 10, c : 5, r/5);
```

2.4 The Module Concept

Standard PASCAL basically assumes that a program consists of a single program text which must be prepared completely before it can be compiled and executed. In many cases, it is more convenient to prepare a program in several parts, called modules, which can then be developed and compiled independently of each other. Moreover, various other programs may use the components of a module without their having to be copied into the source code and recompiled.

For this purpose, a module concept has been introduced into PASCAL–XSC. This new concept offers the possibilities of

- modular programming
- syntax check and semantic analysis beyond the bounds of modules
- implementation of arithmetic packages as standard modules

A module is introduced by the keyword **module** followed by a name and a semicolon. Its body is quite similar in structure to that of a normal program with the exception that the word symbol **global** can be used directly in front of the keywords **const**, **type**, **var**, **procedure**, **function**, and **operator** and directly after **use** and the equal sign in type declarations.

Thus it is possible to declare private types as well as non-private types. The internal structure of a private type is not known outside the declaring module. Objects of such a private type can only be used and manipulated via the procedures, functions and operators supplied by the declaring module.

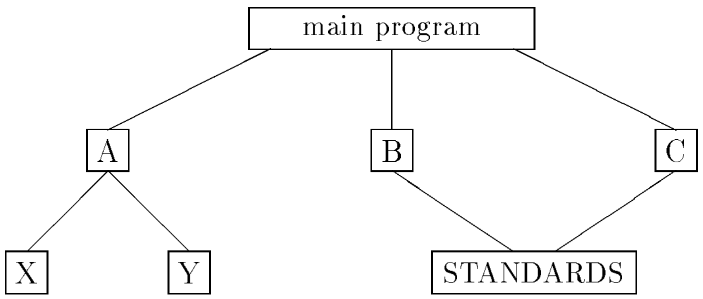
For importing modules with **use** or **use global** the following transitivity rules hold

$$\text{M1 } \mathbf{use} \text{ M2} \quad \text{and} \quad \text{M2 } \mathbf{use\ global} \text{ M3} \;\Rightarrow\; \text{M1 } \mathbf{use} \text{ M3},$$

but

$$\text{M1 } \mathbf{use} \text{ M2} \quad \text{and} \quad \text{M2 } \mathbf{use} \text{ M3} \;\not\Rightarrow\; \text{M1 } \mathbf{use} \text{ M3}.$$

Example: Let a module hierarchy be built up by



All global objects of the modules A, B, and C are visible in the main program unit, but there is no access to the global objects of X, Y and STANDARDS from the main program.

2.5 Dynamic Arrays

In standard PASCAL there is no way to declare dynamic types or variables. The only way to manage memory dynamically in standard PASCAL is through the allocation and deallocation of fixed-size objects which are referred by pointers.

For instance, program packages with vector and matrix operations are typically implemented with fixed (maximum) dimensions. For this reason, only part of the allocated memory is used if the user wants to solve problems with lower dimensions. The concept of dynamic arrays removes this limitation. In particular, the new concept can be described by the following characteristics:

- Dynamics within procedures and functions
- Automatic allocation and deallocation of local dynamic variables
- Economical employment of storage space
- Row access and column access to dynamic arrays
- Compatibility of static and dynamic arrays

Dynamic arrays must be marked with the word symbol **dynamic**. The great disadvantage of the *conformant array* schemes available in standard PASCAL is that they can only be used for parameters and not for variables or function results. So, this standard feature is not fully dynamic.

In PASCAL–XSC, dynamic and static arrays can be used in a very similar manner. For example, a two-dimensional dynamic array type can be declared in the following form:

```
type matrix = dynamic array [*,*] of real;
```

It is also possible to define different dynamic types with corresponding syntactical structures. For example, it might be useful in some situations to identify the coefficients of a polynomial with the components of a vector or vice versa. Since PASCAL is a strictly type-oriented language, such structurally equivalent arrays may only be combined if their types have been previously adapted. The following example shows the definition of a polynomial and of a vector type (note that the type converting functions *polynomial(...)* and *vector(...)* are defined implicitly). Access to the lower and upper index bounds of each dimension is made possible by the new standard functions *lbound(...)* and *ubound(...)* or their abbreviations *lb(...)* and *ub(...)*.

```
type vector = dynamic array [*] of real;

type polynomial = dynamic array [*] of real;

operator + (a, b: vector) res: vector[lb(a)..ub(a)];
var i : integer;
begin
  for i := lb(a) to ub(a)
    res[i] := a[i] + b[lb(b) + i - lb(a)]
end;

var
  v : vector[1..n];
```

```
    p : polynomial[0..n-1];

  ...

    v := vector(p);
    p := polynomial(v);
    v := v + v;
    v := vector(p) + v;  { but not v := p + v; }
```

In addition to accessing each component variable, PASCAL–XSC offers the possibility of accessing subarrays. If a component variable contains an $*$ or a range instead of an index expression, it refers to the subarray with the entire or specified index range in the corresponding dimension. For example, `M[1..2,j]` is the array consisting of the 1st and 2nd elements of the j-th column of a two-dimensional array `M`.

This example demonstrates access to rows or columns of dynamic arrays:

```
  type vector = dynamic array [*] of real;

  type matrix = dynamic array [*] of vector;

  ...

  var
    v : vector[1..n];
    m : matrix[1..n,1..n];

  ...

    v := m[i];
    m[i] := vector(m[*,j]);
```

In the first assignment it is not necessary to use a type converting function since both the left and the right side are of *known dynamic* type. A different case is demonstrated in the second assignment. The left-hand side is of *known dynamic* type, but the right-hand side is of *anonymous dynamic* type, so it is necessary to use the intrinsic converting function *vector(...)*.

2.6 Flexible Arrays

Current releases (Version 3.0 and higher) of PASCAL–XSC include the concept of *flexible* arrays. A dynamic array is called flexible if it can be reallocated with new index bounds and new size at any time during its lifetime.

There are two possibilities of declaring flexible array types. The first possibility is the usual declaration of a dynamic array type, i.e. every dynamic array is also a flexible array.

```
  type vector = dynamic array [*] of real;
```

The second possibility is to provide default index ranges for flexible array variables (for example in a program using many array variables with identical index bounds and only a few variables of the same type with other index bounds). The extended rules for array declaration allow

```
  type vec = dynamic array [1..10] of real;
```

or

```
    type vec = vector [1..10];
```

alternatively.

Now, there are several possibilities to declare flexible array variables:

- We can specify index ranges by

  ```
      var v : vector [1..10];
  ```

  or by

  ```
      var v : dynamic array [1..10] of real;
  ```

  and the variables declared in this way are automatically allocated and deallocated on entry and exit of the subroutine they belong to. But those variables may also be reallocated during the execution of the subroutine they belong to.

- We can omit index ranges by

  ```
      var v : vector;
  ```

  or by

  ```
      var v : dynamic array [*] of real;
  ```

  and we must explicitely allocate `v` by ourselves. Nevertheless, it will be automatically deallocated on exit of the subroutine it belongs to.

- We can use a flexible array with default index ranges by

  ```
      var v : vec;
  ```

  or by

  ```
      var v : vec [1..20];
  ```

  where in both cases automatic memory allocation and deallocation will be carried out, but the allocation process is different. In the first case the index bounds for `v` are the the default index bounds of `vec`, in the second case the specified index bounds replace the default bounds.

Of course, PASCAL–XSC supplies routines for the memory management of dynamic (flexible) arrays. Procedure *allocate* allows the explicit allocation of a dynamic array with specified index bounds. Procedure *free* allows the deallocation of a variable, i.e. the freeing of the memory occupied by a dynamic array variable. Moreover, since access to an array might result in a runtime error, PASCAL–XSC provides the boolean function *allocated* for testing the accessibility of dynamic arrays.

In the following example, we use a real vector in different lengths to perform computations until a desired accuracy is achieved.

```
type
  vector = dynamic array [*] of real;

procedure high_accuracy (basic_length: integer; result: real);
var
  accurate : boolean;
  rvec     : vector;
  k        : integer;
begin
  accurate := false;   k := 0;

  repeat
    k:= k+1;
    allocate (rvec, 1..k*basic_length);

    ...  { computations }

    accurate := ...

    free (rvec);   { might be ommitted }
  until accurate;

  result := ...
end;
```

In our second example, we give a routine for reading integer vectors from a text file, where each vector is preceded by the number of its components. Without the possibility of reallocation of dynamic arrays the solution of this problem is very laborious.

```
type
  vector = dynamic array [*] of integer;

procedure read (var f: text; var v: vector);
var
  i, length : integer;
begin
  if allocated(v) then
    free (v);
  read(f, length);
  allocate(v, 1..length);
  for i:=lb(v) to ub(v) do
    read(f, v[i]);
end;
```

A detailed description of syntax and semantic for the concept of dynamic and flexible arrays is given in Appendix B.

2.7 Accurate Expressions

The theory of computer arithmetic (see [25]) requires the implementation of the dot product with only one rounding according to the following definition (see [6]):

> Given two vectors x and y with n floating-point components each, and a prescribed rounding mode $\Box$, the floating-point result s of the dot product operation (applied to x and y) is defined by
>
> $$s := \Box(\overline{s}) := \Box(x \cdot y) = \Box(\sum_{i=1}^{n} x_i * y_i), \qquad n \geq 1$$

where all arithmetic operations are mathematically exact. Thus s shall be computed as if an intermediate result $\overline{s}$ correct to infinite precision and with unbounded exponent range were first produced and then rounded to the desired floating-point destination format according to the selected rounding mode $\Box$.

Thus the result of the operation must be the exact result of the dot product with just one final rounding applied.

The implementation of enclosure algorithms with automatic result verification or validation (see [14],[17],[18],[27],[32]) makes extensive use of the accurate evaluation of dot products. To evaluate this kind of expression the new datatype *dotprecision* was introduced. Variables of type dotprecision can hold any possible value which results from the evaluation of dot product expressions without loss of accuracy (see [25],[14]). Based upon this type, so-called *accurate expressions* (#-expressions), can be formulated by an accurate symbol (#, #*, #<, #>, or ##) followed by an *exact expression* enclosed in parentheses. The exact expression must have the form of a dot product expression in scalar, vector or matrix structure and is evaluated without any rounding error. Because of this, the result of an *accurate expression* has an error of at most 1 ulp, i.e. at most one unit in the last mantissa place. Tables in the appendix give an overview of possible exact expressions within the accurate expressions (see [16] for the detailed overview).

To obtain the unrounded or correctly rounded result of a dot product expression, the user needs to parenthesize the expression and precede it by the symbol # which may optionally be followed by a symbol for the rounding mode. Table 4 shows the possible rounding modes with respect to the dot product expression form.

Symbol	Expression Form	Rounding Mode	Math. Symbol
#*	scalar, vector or matrix	nearest	$\Box$
#<	scalar, vector or matrix	downwards	$\triangledown$
#>	scalar, vector or matrix	upwards	$\triangle$
##	scalar, vector or matrix	smallest enclosing interval	$\Diamond$
#	scalar only	exact, no rounding	

Table 4: Rounding Modes for Accurate Expressions

In practice, dot product expressions may contain a large number of terms making an explicit notation very cumbersome. To alleviate this difficulty in mathematics, the symbol $\sum$ is used. If for instance A and B are n-dimensional matrices, then the evaluation of

$$d = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

represents a dot product expression. PASCAL–XSC provides the equivalent shorthand notation **sum** for this purpose. The corresponding PASCAL–XSC statement for this expression is

```
d := #(for k:=1 to n sum (A[i,k] * B[k,j]));
```

where d is a *dotprecision* variable.

Dot product expressions or accurate expressions are used mainly in computing a defect (or residual). In the case of a linear system $Ax = b$, $A \in I\!R^{n\times n}$, $x, b \in I\!R^n$, $Ay \approx b$ is considered as an example. Then an enclosure of the defect is given by $\Diamond(b - Ay)$ which in PASCAL–XSC can be realized by means of the dot product expression

```
## (b - A * y);
```

with only one interval rounding operation for each component of the resulting interval vector. To get verified enclosures for linear systems of equations it is necessary to evaluate the defect expression

$$\Diamond(E - RA)$$

where $R \approx A^{-1}$ and E is the identity matrix. In PASCAL–XSC this expression can be programmed as

```
## (id(A) - R * A);
```

where an interval matrix is computed with only one rounding operation per component. The function *id(...)* is defined in the module for real matrix/vector arithmetic and generates an identity matrix of the same shape as its arguments (see section 2.9).

2.8 The String Concept

The tools provided for handling character strings in standard PASCAL do not allow convenient text processing. For this reason, a string concept was integrated into the language definition of PASCAL–XSC which admits a convenient treatment of textual information and, using the operator concept, even symbolic computation. With new data type *string*, the user can work with strings of up to MAXINT characters. When declaring variables of type *string*, the user can specify a maximum string length less than MAXINT. Thus a string s declared by

```
var s : string[40];
```

can be up to 40 characters long. The following standard operations are available:

- concatenation
- actual length
- conversion *string* $\rightarrow$ *real*
- conversion *string* $\rightarrow$ *integer*
- conversion *real* $\rightarrow$ *string*
- conversion *integer* $\rightarrow$ *string*
- extraction of substrings
- position of first appearance
- relational operators <=, <, >=, >, <>, =, and **in**

2.9 Standard Modules

The following standard modules are available:

- interval arithmetic (I_ARI)
- complex arithmetic (C_ARI)
- complex interval arithmetic (CI_ARI)
- real matrix/vector arithmetic (MV_ARI)
- interval matrix/vector arithmetic (MVI_ARI)
- complex matrix/vector arithmetic (MVC_ARI)
- complex interval matrix/vector arithmetic (MVCI_ARI)

These modules may be incorporated via the **use** statement described in section 2.4. As an example, Table 5 exhibits the operators provided by the module for interval matrix/vector arithmetic.

right operand / left operand	integer real	interval	rvector	ivector	rmatrix	imatrix
monadic				+, −		+, −
integer real				*		*
interval			*	*	*	*
rvector		*, /	+*	+*, +, −, *, **in**, =, <>		
ivector	*, /	*, /	+*, +, −, *, =, <>	+*, **, +, −, *, **in**, =, <>, ><, <=, <, >=, >		
rmatrix		*, /		*	+*	+*, +, −, *, **in**, =, <>
imatrix	*, /	*, /	*	*	+*, +, −, *, =, <>	+*, **, +, −, *, **in**, =, <>, ><, <=, <, >=, >

Table 5: Predefined Arithmetic and Relational Operators of the Module MVI_ARI

In addition to these operators, the module MVI_ARI provides the following generically named standard operators, functions, and procedures:

intval, inf, sup, diam, mid, blow, transp, null, id, read, and *write.*

The function *intval* is used to generate interval vectors and matrices, whereas *inf* and *sup* are selection functions for the infimum and supremum of an interval object. The diameter and the midpoint of interval vectors and matrices can be computed by *diam* and *mid*, *blow* yields an interval inflation, and *transp* delivers the transpose of a matrix.

Zero vectors and matrices are generated by the function *null*, while *id* returns an identity matrix of appropriate shape. Finally, there are the generic input/output-procedures *read* and *write*, which may be used in connection with all matrix/vector data types defined in the modules mentioned above.

2.10 Problem-Solving Routines

Routines for solving common numerical problems have been implemented in PASCAL–XSC. The applied methods compute a highly accurate enclosure of the true solution of the problem and, at the same time, prove the existence and the uniqueness of the solution in the computed interval. The advantages of these new routines are :

- The solution is computed with maximum or high, but always controlled accuracy, even in many ill-conditioned cases.
- The correctness of the result is automatically verified, i.e. an enclosing set is computed, which guarantees existence and often also uniqueness of the true solution contained in this set.
- If no solution exists or if the problem is extremely ill-conditioned, an error message is issued.

Among others, PASCAL–XSC routines cover the following subjects:

- linear systems of equations
  - full systems (*real, complex, interval, cinterval*)
  - matrix inversion (*real, complex, interval, cinterval*)
  - least squares problems (*real, complex, interval, cinterval*)
  - computation of pseudo inverses (*real, complex, interval, cinterval*)
  - band matrices (*real*)
  - sparse matrices (*real*)
- polynomial evaluation
  - in one variable (*real, complex, interval, cinterval*)
  - in several variables (*real*)
- zeros of polynomials (*real, complex, interval, cinterval*)
- eigenvalues and eigenvectors
  - symmetric matrices (*real*)
  - arbitrary matrices (*real, complex, interval, cinterval*)
- initial and boundary value problems of ordinary differential equations
  - linear
  - nonlinear

- evaluation of arithmetic expressions
- nonlinear systems of equations
- numerical quadrature
- integral equations
- automatic differentiation
- optimization

3 The Implementation of PASCAL–XSC

The language PASCAL–XSC extends the PASCAL–SC language [7, 8, 28]. Both languages were defined and developed at the Institute of Applied Mathematics at the University of Karlsruhe. The first PASCAL–SC compiler was implemented for Z80 processors in 1980. Because of the small memory of the Zilog machine, an interpreter was used, which slowed down the execution time. This compiler was ported to DOS machines in the early 80's [24]. Three years later a PASCAL–SC compiler generating machine code for Motorola-68000 processors was developed [23]. This system is much faster, but it lacks portability, running only on Motorola-68000 processors. The new PASCAL–XSC system is now available for personal computers, workstations, mainframes, and supercomputers by means of a portable compiler which translates to ANSI C.

The main goal of the system is portability. For that purpose, it is necessary

- to provide easy porting of the compiler and the runtime system
- to avoid the necessity to retarget the compiler for every new computer
- to provide porting of the generated code (cross-compilation)
- to provide consistency of results for all installations

The ANSI C language (as defined in [5]) was chosen as the implementation language and the target language. The main reason for this choice was the extremely wide range of computers for which one or more C compilers are available. Besides the C language allows the programming of portable code. The ANSI C language standard will impel the producers of C compilers to construct the compilers that correspond to the standard and impel them to unify the existing compilers. This makes porting easier. Special compiler options exist to provide cross-compilation. The C language is highly modular. Small overhead for function calls results in high efficiency of the target code.

There are great semantic differences between the PASCAL-XSC and the C language. Since PASCAL-XSC allows dot precision expressions, nested subroutine declarations, overloading of subroutines, dynamic arrays and subarrays, sets and strings, it becomes necessary to simulate these concepts in the target code. Partly this task can be solved using the appropriate functions in the runtime library, but some problems, such as the simulation of nested subroutines, have to be solved inside the compiler.

3.1 Different Real Arithmetics

A special feature of the new compiler is that the basic operations of the real arithmetic are exchangeable to support different applications which may require different properties of the arithmetic (portability, speed or accuracy). See [12] for details.
Supported arithmetics are:

- Software emulation of the IEEE 754 standard arithmetic. A complete floating-point arithmetic for the double format of the IEEE binary floating-point standard [4] is simulated in software. All requirements of the standard are fulfilled including directed roundings, handling of infinity, and exception handling. No special properties of the hardware nor support from the C runtime system are required.
- The hardware arithmetic of the computer in use. The arithmetic operations are supported by the C runtime system. The data format and the accuracy of the operations need not necessarily satisfy the IEEE standard. This arithmetic is intended to be used by programs that shall be "fast".
- Multiple precision arithmetic. It is intended for programs implementing high-precision numerical algorithms. The arithmetic operations are based on the special *multiple precision* data type. Variables of this type may hold values with a varying number of mantissa digits during the execution of the program.
- Decimal arithmetic. The BCD version with decimal real and longreal formats is intended to avoid the conversion errors occurring during input and output of numerical data.
- A user-defined arithmetic. Standard real arithmetic can be replaced by a user-defined real arithmetic in a very simple manner (see [2], [3]). The user must ensure that all features defined for standard real arithmetic will be also available for this new arithmetic.

3.2 The PASCAL–XSC Development System

The PASCAL–XSC system [3] includes:

- The manager.
- The PASCAL–XSC to C compiler.
- The listing generator.
- The runtime library.
- The configuration program.

The main purpose of the manager is to make the program development cycle more user-friendly and to reduce the number of accidental errors. It is achieved by freeing the user from having to supply the information about directory conventions and options of the PASCAL–XSC compiler, C-compiler, and linker in use. The manager offers a "make" facility by linking automatically all the modules that the current program depends on.

The dependencies and connections of modules are completely checked for consistency using interface files associated with the modules that are mentioned in the "use" clause, if any. The number and the types of actual arguments are checked for conformity to the formal parameters. A module may import other modules. In general, the module dependencies in an executable program can be described by a directed acyclic graph.

The compiler offers a comprehensive error-checking facility including lexical, syntactical and semantical checking and error recovery. If a module is changed, it is quite natural that it must be recompiled before the modules which "use" it are compiled. The compiler checks this condition automatically and generates an error message if the time-compatibility of modules is violated. The listing generator is called after compiling a program or module containing errors. It produces a readable listing with error messages and pointers to the exact positions of the errors: the line and the column. It is possible to correct errors by editing the listing. There exists a program that reads the listing and reconstructs the source file from the listing.

After the installation of the compiler, the user may change some system dependencies such as path names and filetype names as well as default values for the compiler options. These modifications are done by means of the *configuration program*.

3.3 The Current State of Implementation

The conformity of the PASCAL–XSC compiler to standard PASCAL [10] was tested using "The PASCAL validation suite" of the Tasmania University [33]. Extensive tests have been carried out concerning different PASCAL–XSC extensions. The system is widely spread and used for educational purposes and for software development.
Until now the PASCAL–XSC system has been successfully installed and thoroughly tested on many computers (see table 6). On some systems hardware arithmetic is supported, making the generated programs faster.

Computer	Operating System	C compiler
PC	MS-DOS/Windows	GNU C++
PC	OS/2 3.0	GNU C++
PC	LINUX	GNU C
HP 9000/700 Series	UNIX	HP C
Sun SPARC Station	SunOS 5.x	SUN C
Sun SPARC Station	SunOS 4.1	Standard C
IBM RS/6000	AIX	ANSI C
Silicon Graphics	IRIX	GNU C
CONVEX C2-C4	UNIX	Convex CC

Table 6: Availability of the PASCAL–XSC System

Along with the commercial versions several free versions of the PASCAL–XSC compiler (for DOS, OS/2, LINUX, etc.) are available. The software and further information can be found on the homepage

`http://www.xsc.de`

of *Numerik Software GmbH* (email: `numerik_software@csi.com`).

4 PASCAL–XSC Sample Programs

In the following, some PASCAL–XSC programs are listed, demonstrating the use of the arithmetic modules and various concepts of PASCAL–XSC.

Well-known algorithms were intentionally chosen so that a brief explanation of the mathematical background will suffice. Since the programs are largely self-explanatory, comments are kept to a minimum.

1. Interval Newton Method
2. Runge-Kutta Method
3. Trace of a Product Matrix
4. Verified Solution of a Linear System

4.1 Interval Newton Method

An inclusion of a zero of the real-valued function $f(x)$ is computed. It is assumed that $f'(x)$ is a continuous function on the interval $[a, b]$, where $0 \notin \{f'(x) : x \in [a, b]\}$ and $f(a) \cdot f(b) < 0$. If an inclusion X_n for the zero of such a function $f(x)$ is already known, a better inclusion X_{n+1} may usually be computed by the iteration formula:

$$X_{n+1} := (m(X_n) - \frac{f(m(X_n))}{f'(X_n)}) \cap X_n \quad ,$$

where $m(X)$ is some point in the interval X (for example the midpoint). For this example, the function $f(x) = \sqrt{x} + (x + 1) \cdot \cos x$ is used. In PASCAL–XSC, interval expressions are written in mathematical notation. Generic function names are used for the interval square root and interval sine and cosine functions. For the mathematical theory, see [1].

```
program  inewt (input, output);
use
  i_ari; { interval arithmetic }
var
  x, y : interval;

{----------------------------------------------------------------------------}

function  f (r: real): interval;
var
  x : interval;
begin
  x := r; { converts r to type interval to obtain a verified inclusion }
  f := sqrt(x) + (x + 1) * cos(x)
end;

{----------------------------------------------------------------------------}

function  deriv (x: interval): interval;
begin
  deriv := 1 / (2 * sqrt(x)) + cos(x) - (x + 1) * sin(x)
end;

{----------------------------------------------------------------------------}
```

```
function  criter (x : interval) : boolean;
begin
  criter := (sup(f(inf(x)) * f(sup(x))) < 0) and not (0 in deriv(x));
end;

{---------------------------------------------------------------------------}

begin { main program }
  { The interval notation for I/O in PASCAL-XSC is  [ inf , sup ] }
  { mid(x) is a function returning the midpoint of the interval x }

  write ('Please enter starting interval : ');  read (y);

  while inf(y) <> sup(y) do
  begin
    if criter(y) then
      repeat
        x := y;
        writeln (x);
        y := ( mid(x) - f(mid(x))/deriv(x) ) ** x;
      until x = y
    else
      writeln ('Criterion not satisfied !');

    writeln;
    write ('Please enter starting interval : ');  read (y);
  end;
end.
```

With the starting interval $[2,3]$ the computed inclusions are

```
[                2.0E+000,                3.0E+000]
[                2.0E+000,                2.3E+000]
[               2.05E+000,               2.07E+000]
[            2.05903E+000,            2.05906E+000]
[     2.059045253413E+000,     2.059045253417E+000]
[  2.059045253415143E+000,  2.059045253415145E+000]
```

4.2 Runge-Kutta Method

The initial-value problem for a system of differential equations is to be solved. The Runge-Kutta method to solve **one** differential equation may be written in standard PASCAL in an almost mathematical notation. In PASCAL–XSC it is possible to use the same notation for a **system** of differential equations. The concept of dynamic arrays is used to make the program independent of the size of the system. Only as much storage as needed is occupied during runtime. The following system of first-order differential equations

$$Y' = F(x, Y)$$

with initial condition $Y(x_0) = Y_0$ is considered. If the solution Y is known at a point x, then the approximation $Y(x+h)$ is computed by

$$\begin{aligned}
K_1 &= h \cdot F(x, Y), \\
K_2 &= h \cdot F(x + h/2, Y + K_1/2), \\
K_3 &= h \cdot F(x + h/2, Y + K_2/2),
\end{aligned}$$

$$\begin{aligned} K_4 &= h \cdot F(x+h, Y+K_3), \\ Y(x+h) &= Y + (K_1 + 2 * K_2 + 2 * K_3 + K_4)/6. \end{aligned}$$

Starting at x_0, an approximate solution may be computed at the points $x_i = x_0 + i \cdot h$. We supply function F in a module.

```
module f;

use
  mv_ari; { matrix/vector arithmetic }

global const
  dim = 3;

{---------------------------------------------------------------------------}

global function F (x: real; y: rvector): rvector[1..dim];
begin
  f[1] := y[1] - y[2];
  f[2] := exp(x) * y[3];
  f[3] := (y[1] - y[2]) / exp(x);
end;

{---------------------------------------------------------------------------}

global procedure init (var x, h: real; var y: rvector);
begin
  x := 0;  h := 0.1;  y[1] := 0;  y[2] := 1;  y[3] := 1
end;

end. { of module f }
```

Using module f, we can write the following program.

```
program  runge (input, output);

use
  mv_ari, f;

var
  i                 : integer;
  x, h              : real;
  y, k1, k2, k3, k4 : rvector[1..dim];

begin
  init(x, h, y);

  { Classical Runge-Kutta method (10 steps) for a system  }
  { of first-order differential equations  y' = F(x, y)   }
  for i:=1 to 10 do
  begin
    k1 := h * f(x, y);
    k2 := h * f(x + h / 2, y + k1 / 2);
    k3 := h * f(x + h / 2, y + k2 / 2);
    k4 := h * f(x + h, y + k3);
    y  := y + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    x  := x + h;
    writeln ('x = ', x);
    writeln ('y = ', y);
  end;
end.
```

4.3 Trace of a Product Matrix

The following PASCAL–XSC program demonstrates the use of *accurate*-expressions. The trace of a product matrix $A \cdot B$ is computed without evaluating the product matrix itself. The result will be of maximum accuracy, i.e. it is the best possible floating-point approximation of the exact solution. The trace of the product matrix is given by

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \cdot b_{ji}.$$

A corresponding program is

```
program  trace (input, output);

use
  mv_ari; { matrix/vector arithmetic }

var
  n : integer;

{----------------------------------------------------------------------------}

procedure main (n: integer);

var
  i, j : integer;
  s, d : real;
  A, B : rmatrix [1..n,1..n];

begin
  read (A, B);
  s := 0;
  for i:= 1 to n do
    s := s + A[i] * rvector(B[*,i]);
  writeln ( 'Trace of A*B computed with scalar product :', s);

  d := #*( for i:=1 to n sum( A[i] * rvector(B[*,i]) ));
  writeln ( 'Trace of A*B computed with #-expression   :', d);
end;

{---------------------------------------------------------------------------}

begin
  read(n);  main(n);
end.
```

With the following starting matrices

$$A = \begin{pmatrix} 1e9 & 8 & 126 & -237 \\ 100 & 2 & -12 & 1 \\ 1e5 & 10 & -1e7 & 81 \\ 13 & -3 & 30 & 1e{-7} \end{pmatrix}$$

$$B = \begin{pmatrix} 1e8 & 85 & 8 & 6 \\ 12 & 3 & 1e3 & 156 \\ 3 & 14 & 1e10 & 13 \\ 2 & -8332 & -1e4 & -1e{-8} \end{pmatrix}$$

the computed results are

```
Trace of A*B computed with scalar product : -9.999999999999999E-016
Trace of A*B computed with #-expression   :  5.999999999999999E+000
```

4.4 Verified Solution of a Linear System of Equations

The example demonstrates a program for the verified solution of a system of linear equations. The program delivers either a verified solution or a corresponding failure message.

Employing the module LIN_SOLV, the solution of a system of linear equations is enclosed in an interval vector by successive interval iterations.

The procedure *main*, which is called in the body of *lin_sys*, is only used for reading the dimension of the system and for allocation of the dynamic variables. The numerical method itself is started by the call to procedure *linear_system_solver* defined in module *lin_solv*. This procedure may be called with arrays of arbitrary but matching dimension.

For detailed information on iteration methods with automatic result verification, see [14], [17], [18], [27], or [31], for example.

```
module lin_solv;

use i_ari,   { interval arithmetic               }
    mv_ari,  { matrix/vector arithmetic          }
    mvi_ari; { matrix/vector interval arithmetic }

{---------------------------------------------------------------------------}

priority inflated = *; { priority level 2 }

{---------------------------------------------------------------------------}

operator inflated (a: ivector; eps: real) infl: ivector[1..ub(a)];

{ Computes the so-called epsilon inflation of an interval vector. }

var
  i : integer;
  x : interval;

begin
  for i:= 1 to ub(a) do
  begin
    x := a[i];
    if (diam(x) <> 0) then
      a[i] := (1+eps)*x - eps*x
    else
      a[i] := intval( pred (inf(x)), succ (sup(x)) );
  end; {for}
  infl := a;
end;  { operator inflated }

{---------------------------------------------------------------------------}

function approximate_inverse (A: rmatrix): rmatrix[1..ub(A),1..ub(A)];

{ Computation of an approximate inverse of the (n,n)-matrix A }
{ by application of the Gaussian elimination method.          }

var
  i, j, k, n :  integer;
  factor      :  real;
  R, Inv, E   :  rmatrix[1..ub(A),1..ub(A)];

begin
  n := ub(A);  { dimension of A }
  E := id(E);  { identity matrix }
  R := A;

  { Gaussian elimination step with unit vectors as    }
  { right-hand sides. Division by R[i,i]=0 indicates  }
  { that matrix A is probably singular .                      }

  for i:= 1 to n do
    for j:= (i+1) to n do
    begin
      factor := R[j,i]/R[i,i];
      for k:= i to n do
        R[j,k] := #*(R[j,k] - factor*R[i,k]);
      E[j] := E[j] - factor*E[i];
    end;  { for j:= ... }

  { Backward substitution delivers the rows of the inverse of A.  }

  for i:= n downto 1 do
    Inv[i] := #*(E[i] - for k:= (i+1) to n sum(R[i,k]*Inv[k]))/R[i,i];

  approximate_inverse := Inv;
end;  { function approximate_inverse }

{-----------------------------------------------------------------------------}

global procedure linear_system_solver (A: rmatrix; b: rvector;
                                       var x: ivector; var ok: boolean);

{ Computation of a verified enclosure vector for the solution of the  }
{ linear system of equations. If an enclosure is not achieved after   }
{ a certain number of iteration steps, the algorithm is stopped and   }
{ the parameter ok is set to false.                                   }

const
  epsilon   =  0.25; { Constant for the epsilon inflation }
  max_steps =  10;   { Maximum number of iteration steps }

var
  i    :  integer;
  y, z :  ivector[1..ub(A)];
  R    :  rmatrix[1..ub(A),1..ub(A)];
  C    :  imatrix[1..ub(A),1..ub(A)];

begin
  R := approximate_inverse(A);

  { R*b is an approximate solution of the linear system    }
  { and z is an enclosure of this vector. However, it does }
  { not usually enclose the true solution.                 }

  z := R * intval(b);

  { An enclosure of I - R*A is computed with maximum accuracy.         }
  { The (n,n) identity matrix is generated by the function call id(A). }

  C := ##(id(A) - R*A);
  x := z;  i := 0;
  repeat
    i := i + 1;
    y := x inf/lated epsilon; { To obtain a true enclosure, the interval }
                              { vector c is slightly enlarged.          }
    x := z + C*y;             { The new iterate is computed.            }
   ok := x in y;              { Is c contained in the interior of y?    }
  until ok or (i = max_steps);
end;  { procedure linear_system_solver }

{---------------------------------------------------------------------------}

end.  { module lin_solv }
```

The following program can be used to apply the routine supplied by module *lin_solv*.

```
program lin_sys (input, output);

use lin_solv, { linear system solver              }
    mv_ari,   { matrix/vector arithmetic          }
    mvi_ari;  { matrix/vector interval arithmetic }

var n  :  integer;

{---------------------------------------------------------------------------}

procedure main (n : integer);

{ The matrix A and the vectors b, x are allocated dynamically with   }
{ this subroutine being called. The matrix A and the right-hand side }
{ b are read in and linear_system_solver is called.                  }

var
  ok :  boolean;
  b  :  rvector[1..n];
  x  :  ivector[1..n];
  A  :  rmatrix[1..n,1..n];

begin
  writeln('Please enter the matrix A:');
  read(A);
  writeln('Please enter the right-hand side b:');
  read(b);

  linear_system_solver(A,b,x,ok);

  if ok then
    begin
      writeln('The given matrix A is non-singular and the solution ');
      writeln('of the linear system is contained in:');
      write(x);
    end
  else
    writeln('No solution found !');
end;  { procedure main }

{---------------------------------------------------------------------------}

begin
  write('Please enter the dimension n of the linear system: ');
  read(n);
  main(n);
end. { program lin_sys }
```

Appendix

A Review of #-Expressions

A.1 Real and Complex #-Expressions

Syntax: #-Symbol (Exact Expression)

#-Symbol	Result Type	Summands Permitted in the Exact Expression
#	*dotprecision*	• variables, constants, and special function calls of type *integer*, *real*, or *dotprecision* • products of type *integer* or *real* • scalar products of type *real*
#* #< #>	*real*	• variables, constants, and special function calls of type *integer*, *real*, or *dotprecision* • products of type *integer* or *real* • scalar products of type *real*
	complex	• variables, constants, and special function calls of type *integer*, *real*, *complex*, or *dotprecision* • products of type *integer*, *real*, or *complex* • scalar products of type *real* or *complex*
	rvector	• variables and special function calls of type *rvector* • products of type *rvector* (e.g. *rmatrix* ∗ *rvector*, *real* ∗ *rvector* etc.)
	cvector	• variables and special function calls of type *rvector* or *cvector* • products of type *rvector* or *cvector* (e.g. *cmatrix* ∗ *rvector*, *real* ∗ *cvector* etc.)
	rmatrix	• variables and special function calls of type *rmatrix* • products of type *rmatrix*
	cmatrix	• variables and special function calls of type *rmatrix* or *cmatrix* • products of type *rmatrix* or *cmatrix*

A.2 Real and Complex Interval #-Expressions

Syntax: ## (Exact Expression)

#-Symbol	Result Type	Summands Permitted in the Exact Expression
##	*interval*	• variables, constants, and special function calls of type *integer*, *real*, *interval*, or *dotprecision* • products of type *integer*, *real*, or *interval* • scalar products of type *real* or *interval*
	cinterval	• variables, constants, and special function calls of type *integer*, *real*, *complex*, *interval*, *cinterval*, or *dotprecision* • products of type *integer*, *real*, *complex*, *interval*, or *cinterval* • scalar products of type *real*, *complex*, *interval*, or *cinterval*
	ivector	• variables and special function calls of type *rvector* or *ivector* • products of type *rvector* or *ivector*
	civector	• variables and special function calls of type *rvector*, *cvector*, *ivector*, or *civector* • products of type *rvector*, *cvector*, *ivector*, or *civector*
	imatrix	• variables and special function calls of type *rmatrix* or *imatrix* • products of type *rmatrix* or *imatrix*
	cimatrix	• variables and special function calls of type *rmatrix*, *cmatrix*, *imatrix*, or *cimatrix* • products of type *rmatrix*, *cmatrix*, *imatrix*, or *cimatrix*

B Dynamic and Flexible Arrays – A Language Reference Supplement

This section describes the current concept of dynamic and flexible arrays, which is not part of early compiler versions (< 3.0). First, a summary of the basic concept (see [22]) is given. Then, the new features are discussed. Since this section is intended to be a supplement to the Language Reference [22], we use the notation from [22] for describing the syntax of the new constructs. It is a simplified Backus-Naur-form which looks similar to usual program code. Syntax descriptions are marked by a vertical black bar at the left margin.

B.1 Dynamic Arrays

The basic characteristic of this concept is the possibility of using dynamic entities within subroutines. Locally declared dynamic array variables are automatically allocated and deallocated during the execution of the subroutine they belong to. Moreover, it is possible to access subarrays.

The type declaration for a dynamic array is similar to the declaration of a static array type. One only has to insert the keyword **dynamic** and to replace the index ranges by asterisks.

Example: Type declaration for a real vector.

```
type vector = dynamic array [*] of real;
```

The index ranges can be declared individually for each dynamic array variable.

Example: Declaration of a real vector variable.

```
var v : vector [1..10];
```

The main application of dynamic arrays is their use within subroutines. Consider the following schematic example of the procedure `do_something`:

```
procedure do_something (n: integer);
var
  local : vector [1..n];
begin
  ... { do something }
end;
```

Here, the procedure is declared with an integer parameter `n`. By means of this parameter the index bounds of the local variable `local` are specified. This means, that `local` may hold a different number of elements upon different calls of `do_something`. The disadvantage of this method is that it is not possible to reallocate `local` while the procedure is being executed. In practice, however, it is desirable to be able to use a structured variable[1] with a different number of elements for different purposes within the same subroutine. This could be realized by declaring several dynamic array variables with different element numbers. However, to minimize memory usage it should be possible to reuse variables, i.e. to reallocate them, whenever this is appropriate. This is the motivation for the extension to the concept of dynamic arrays we discuss in the next section.

[1]In our example this is a real vector

B.2 Flexible Arrays

We call an array *flexible* if it can be reallocated with new index bounds and new size at any time during its lifetime.

The realization of this concept led to the following basic ideas:

- The syntax and semantics for declaring dynamic array types and dynamic array variables is extended.
- The use of flexible arrays according to the previous rules for dynamic arrays does not result in different behaviour of PASCAL–XSC programs.
- Standard procedures for memory allocation and deallocation for flexible arrays are provided.
- Assignment of a flexible array to another flexible array implicitly allocates the destination array, if it has not been allocated before.
- The semantics of type declarations is extended: flexible arrays may be used as components of other composite data types.

B.2.1 Declaring a Flexible Array Type

The syntax for specifying a flexible array type (FlexTypeSpecification) in a type declaration is as follows:

dynamic array [DimensionList] **of** TypeIdentifier

A DimensionList is either a list[2] of asterisks (*) or a list of index types. An index type is specified by either the type identifier of an integer subrange type or by explicitly specifying the index bounds in the usual way:

IntegerExpression .. IntegerExpression

Thus, there are two possibilities of declaring flexible array types. The first possibility simply is the adaption of the old declaration rules,

```
type vector = dynamic array [*] of real;
```

i.e. every dynamic array is also a flexible array. The second possibility is to provide default index ranges by

```
type vec = dynamic array [1..10] of real;
```

or by

```
type vec = vector [1..10];
```

for flexible array variables. In practice, a program often uses many array variables of the same array type with identical index bounds and only a few variables of the same type but with other index bounds. For this situation, the extended rules for array declaration allow the specification of default index ranges.

When declaring flexible array types with more than one index ranges it is not allowed to mix asterisks with default index ranges. Thus the semantics for using flexible arrays is kept simple.

[2]A list always consist of at least one element. If more than one element is to be specified then the elements have to be separated by commas.

B.2.2 Declaring Flexible Array Variables

The syntax for the declaration of a dynamic array variable has been extended according to the changes for type declarations in Section B.2.1. It is possible to use a type identifier by

> **var** IdentifierList : FlexTypeIdentifier
> [DimensionList] { may be omitted }

or an explicit type specification by

> **var** IdentifierList : FlexTypeSpecification

Again, these rules allow several possibilities of declaring flexible array variables. With the type identifiers from Section B.2.1, we can

- specify index ranges by

  ```
  var v : vector [1..10];
  ```

  or by

  ```
  var v : dynamic array [1..10] of real;
  ```

  and the variables declared in this way are automatically allocated and deallocated on entry and exit of the subroutine they belong to. But those variables may also be reallocated during the execution of the subroutine they belong to.

- omit index ranges by

  ```
  var v : vector;
  ```

  or by

  ```
  var v : dynamic array [*] of real;
  ```

  and we must explicitly allocate `v` by ourselves. Nevertheless, it will be automatically deallocated on exit of the subroutine it belongs to.

- use a flexible array with default index range by

  ```
  var v : vec;
  ```

  or by

  ```
  var v : vec [1..20];
  ```

  where in both cases automatic memory allocation and deallocation will be carried out, but the allocation process is different. In the first case the index bounds for `v` are the the default index bounds of `vec`, in the second case the specified index bounds replace the default bounds.

The variable `v` may be reallocated by the user in any case.

The concept of flexible arrays is an extension to that of dynamic arrays. Programs developed by exclusively using the previous dynamic array features can be compiled with current (flexible) compiler versions without change.

In the following descriptions of further language extensions it is no longer necessary to distinguish between dynamic and flexible arrays. These features apply to both. Consequently we only speak of dynamic arrays from now on.

B.2.3 Memory Management Subroutines

This section introduces a set of subroutines for handling dynamic arrays, in particular for memory management. The first routine is the procedure *allocate* which allows the explicit allocation of a dynamic array with the specified index bounds. It is called in the form:

allocate (DynamicArrayVariable , IndexRangesList);

Its first parameter is a dynamic array variable which is followed by a list of index ranges. Index ranges are specified in the usual way by specifying the index bounds:

IntegerExpression .. IntegerExpression

The number of index ranges specified must be the same as the number of index ranges in the declaration of the corresponding dynamic array type of the first parameter. If the array variable passed to *allocate* is not allocated yet, it will be allocated with the specified index bounds. If an array variable is already allocated, it first will be deallocated. The latter allows the user to reallocate an array with new index bounds and new size.

A further procedure provides the possibility of freeing memory occupied by a dynamic array variable, i.e. a dynamic array may explicitly be deallocated. It is called by

free (DynamicArrayVariable);

free has only a single parameter which must be a dynamic array variable. After a call of *free* the index bounds of the array parameter are undefined as long as the array is not allocated again. *free* will have no effect if an array is not allocated when it is passed as a parameter.

Access to an array which is not allocated may have undesirable consequences such as a runtime error (see B.2.4). Therefore, PASCAL–XSC provides a function for testing the accessibility of dynamic arrays. It is called by

allocated (DynamicArrayVariable)

and delivers a boolean result. *allocated* yields the value *true*, if the dynamic array variable which has to be passed as the only parameter is allocated, and *false* otherwise.

Example: In Section B.1, procedure `do_something` was an example of how to use dynamic arrays within subroutines. It contained a declaration of the local variable `local` the size of which was specified by the integer parameter of the procedure.

Now, let us assume that we want to double the size of `local` within the procedure. By means of the new standard subroutines described in Section B.2.3, we may change `do_something` as follows:

```
type vector = dynamic array [*] of real;

procedure do_something (n: integer);
var
  local : vector;
begin
  allocate (local, 1..n);
  ...  { do something }
  free (local);  { might be omitted }
  allocate (local, 1..2*n);
  ...  { do something else }
end;
```

The first call to *allocate* sets up the variable local with indices ranging from 1 to `n`. After some further statements we deallocate `local` by calling *free*. In this example, this would not be necessary because the following call of *allocate* first deallocates `local` automatically. However, if the program was in danger to run out of memory and `local` was not be used in the remaining statements of the procedure, it would surely be useful to deallocate the array here. Finally, the second call to *allocate* sets up `local` once more, this time with an index range from 1 to 2`n`.

B.2.4 Access to and Assignment of Dynamic Arrays

Compared to the present implementation (see [22]), the semantics of an assignment statement

```
A := B;
```

where A and B are assignment compatible dynamic arrays, will not change with the following exceptions.

1. A runtime error will be issued if `B` is not allocated.
2. if `A` is not allocated, it will be allocated before assignment will be carried out. `A` will have the same size and index bounds as `B`.

A runtime error is issued if a dynamic array which is not allocated is accessed in an expression and if array indices are to be checked. Otherwise, if array indices are not checked then the effect of accessing a dynamic array which is not allocated is undefined.[3]

[3]The checking of array indices is controlled by compiler options.

B.2.5 Dynamic Arrays as Components of Other Types

Dynamic arrays may now be components of other structured data types like static arrays or records. They may also be referenced by pointers. This is not a syntax extension but an extension to the semantics of type declarations. The declaration of a dynamic array type as the component type of a composite data type follows the rules for dynamic array type declarations given in Section B.2.1. In particular, default index ranges may be specified.

Examples:

```
type
  rec = record
          a, b : real;
          v    : dynamic array [*] of real
        end;

type
  rec = record
          a, b : real;
          v    : dynamic array [1..5] of real
        end;
```

The rules for allocation and deallocation of dynamic components are the same as for dynamic array variables.

Special care has to be taken when a program uses pointers to dynamic arrays. Consider the following

Example:

```
type
  dyn  = dynamic array [*] of integer;
  dyn2 = dynamic array [1..10] of integer;

var
  p1 = ↑dyn;
  p2 = ↑dyn2;

begin
  new (p2);               { automatic allocation of p2↑ }

  new (p1);               { allocation of a runtime descriptor only }

  allocate (p1↑, 1..10);  { explicit allocation of p1↑ }

  ...                     { do something }
end.
```

As you can see `p2` points to a dynamic array with default index range from 1 to 10. Therefore, the call of *new* automatically allocates the dynamic array `p2` points to. This is not the case for the second call of *new*. Since `p1` points to a dynamic array which has no default index ranges *new* only creates a runtime descriptor for the array. The array itself has to be explicitly allocated with a call of *allocate*.

References

[1] Alefeld, G. and Herzberger, J.: *Introduction to Interval Analysis.* Academic Press, 1983.

[2] Allendörfer, U. and Shiriaev, D.: *PASCAL–XSC to C — A portable PASCAL–XSC compiler.* In [19], 91–104, 1991.

[3] Allendörfer, U. and Shiriaev, D.: *PASCAL–XSC — A portable development system.* In [9], 1992.

[4] American National Standards Institute / Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std. 754-1985, New York, 1985.

[5] American National Standard for Information Systems: *Programming Language C.* ANSI X3.159-1989.

[6] Bohlender, G., Cordes, D., Knöfel, A., Kulisch, U., Lohner, R., and Walter, W. V.: *Proposal for Accurate Floating-Point Vector Arithmetic.* In Adams, E. and Kulisch, U. (Eds.): *Scientific Computing with Automatic Result Verification*, 87–102, Academic Press, Orlando, to be published 1992.

[7] Bohlender, G., Rall, L., Ullrich, Ch., and Wolff von Gudenberg, J.: *PASCAL-SC: A Computer Language for Scientific Computation.* Academic Press, New York, 1987.

[8] Bohlender, G., Rall, L., Ullrich, Ch. and Wolff von Gudenberg, J.: *PASCAL-SC – Wirkungsvoll programmieren, kontrolliert rechnen.* Bibliographisches Institut, Mannheim, 1986.

[9] Brezinsky, C. and Kulisch, U., (Eds): *Computational and Applied Mathematics I — Algorithms and Theory.* Proceedings of the 13th IMACS World Congress, Dublin, Ireland. Elsevier, Science publishers B.V., to be published in 1992.

[10] British Standards Institute: *Computer programming language PASCAL.* BS 6192:1982.

[11] Buchholz, W.: *The IBM System/370 Vector Architecture.* IBM Systems Journal 25/1, 1986.

[12] Cordes, D.: *Runtime System for a PASCAL–XSC Compiler.* In [19], 151–160, 1991.

[13] Däßler, K. and Sommer, M.: *PASCAL, Einführung in die Sprache.* Norm Entwurf DIN 66256, Erläuterungen. Springer-Verlag, Berlin, 1983.

[14] Geörg, S., Hammer, R., Kulisch, U., Ratz, D. (Hrsg.): *Wissenschaftliches Rechnen mit Ergebnisverifikation – Eine Einführung.* Akademie Verlag, Ost-Berlin, Vieweg, Wiesbaden, 1989.

[15] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In [32], 1990.

[16] Hammer, R.: *Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen.* Dissertation, Universität Karlsruhe, 1992.

[17] Hammer, R., Hocks, M., Kulisch, U., and Ratz, D.: *Numerical Toolbox for Verified Computing I*, Springer-Verlag, Berlin, 1993.

[18] Kaucher, E., Kulisch, U., and Ullrich, Ch. (Eds.): *Computer Arithmetic – Scientific Computation and Programming Languages.* Teubner, Stuttgart, 1987.

[19] Kaucher, E., Markov, S. M., and Mayer, G. (Eds): *Computer Arithmetic, Scientific Computation and Mathematical Modelling.* IMACS Annals on Computing and Applied Mathematics **12**, J.C. Baltzer, Basel, 1991.

[20] Kirchner, R. and Kulisch, U.: *Accurate Arithmetic for Vector Processors.* Journal of Parallel and Distributed Computing **5**, 250-270, 1988.

[21] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch.: *PASCAL–XSC Sprachbeschreibung mit Beispielen.* Springer-Verlag, Heidelberg, 1991.

[22] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch.: *PASCAL–XSC Language Reference with Examples.* Springer-Verlag, Heidelberg, 1992.

[23] Kulisch, U. (Ed.): *PASCAL-SC: A PASCAL Extension for Scientific Computation*, Information Manual and Floppy Disks, Version ATARI ST. Teubner, Stuttgart, 1987.

[24] Kulisch, U. (Ed.): *PASCAL-SC: A PASCAL Extension for Scientific Computation*, Information Manual and Floppy Disks, Version IBM PC/AT (DOS). Teubner, Stuttgart, 1987.

[25] Kulisch, U. and Miranker, W. L.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[26] Kulisch, U. and Miranker, W. L. (Eds.): *A New Approach to Scientific Computation.* Academic Press, New York, 1983.

[27] Kulisch, U. and Stetter, H. J. (Eds.): *Scientific Computation with Automatic Result Verification.* Computing Suppl. **6**, Springer-Verlag, Wien, 1988.

[28] Neaga, M.: *Erweiterungen von Programmiersprachen für wissenschaftliches Rechnen und Erörterung einer Implementierung.* Dissertation, Universität Kaiserslautern, 1984.

[29] Neaga, M.: *PASCAL-SC – Eine PASCAL-Erweiterung für wissenschaftliches Rechnen.* In: [14], 1989.

[30] Ratz, D.: *The Effects of the Arithmetic of Vector Computers on Basic Numerical Methods.* In: [32], 1990.

[31] Rump, S. M.: *Solving Algebraic Problems with High Accuracy.* In [26], 1983.

[32] Ullrich, Ch. (Ed.): *Contributions to Computer Arithmetic and Self-Validating Numerical Methods.* J. C. Baltzer AG, Scientific Publishing Co., IMACS, 1990.

[33] Wichmann, B. and Ciechanowicz, Z.J. (Eds.): PASCAL Compiler Validation. John Wiley & Sons, 1983.