PASCAL–XSC

A short introduction to the new language

Numerik Software GmbH
Rettigstr. 6
W-7570 Baden-Baden
Federal Republic of Germany
Tel: +49 (721) 370913
Fax: +49 (721) 370928
email: ae18@dkauni2.bitnet

PASCAL–XSC
A short introduction to the new language

Abstract

The new programming language PASCAL–XSC is presented with an emphasis on the new concepts for scientific computation and numerical data processing of the PASCAL–XSC compiler. PASCAL–XSC is a universal PASCAL extension with extensive standard modules for scientific computation. It is available for personal computers, workstations, mainframes and supercomputers by means of an implementation in C.

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

The most important advantage of the new language is that programs written in PASCAL–XSC are easily readable. This is due to the fact that all operations, even those in the higher mathematical spaces, have been realized as operators and can be used in conventional mathematical notation.

In addition to PASCAL–XSC a large number of numerical problem-solving routines with automatic result verification are available. The language supports the development of such routines.

1 Introduction

These days, the elementary arithmetic operations of electronic computers are usually approximated by floating-point operations of highest accuracy. In particular, this means that for any choice of operands, the computed result coincides with the rounded exact result of the operation. See the IEEE Arithmetic Standard [1] as an example. This arithmetical standard also requires the four basic arithmetic operations $+, -, *$, and / with directed roundings. A large number of processors already on the market provide these operations. So far, however, no common programming language allows 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 super-computers provide additional arithmetic operations such as “multiply and add” and “accumulate” or “multiply and accumulate” (see [7]). 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 a scalar processor (see [9],[25]).

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

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 accessable in the language. Further arithmetic operations for intervals and complex numbers and even vector/matrix operations provided by precompiled arithmetical modules are defined with maximum accuracy according to the rules of semimorphism (see [19]).

2 The Language PASCAL–XSC

PASCAL–XSC is an eXtension of the programming language PASCAL for Scientific Computation. A first approach to such an extension (PASCAL–SC) has been available since 1980. The specification of the extensions has been continuously improved in recent years by means of essential language concepts, and the new langauge PASCAL–XSC [14],[15] was developed. It is now available for personal computers, workstations, mainframes, and supercomputers by means of an implementation in C. PASCAL–XSC contains the following features:

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

- Exact evaluation of expressions (#-expressions)

The new language features, developed as an extension of PASCAL, will be discussed in the following sections.

2.1 Standard Data Types, Predefined Operators, and Functions

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

interval	*complex*	*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 theses types in the arithmetic modules of PASCAL–XSC (see section 2.8). All of these operators deliver results with maximum accuracy. In table 1 the 29 predefined standard operators of PASCAL–XSC are listed according to priority.

Type	Priority	Operators
monadic	3 (highest)	monadic +, monadic −, **not**
multiplicative	2	**and**, **div**, **mod** *, *<, *>, /, /<, />, **
additive	1	**or** +, +<, +>, −, −<, −>, +*
relational	0 (lowest)	**in** =, <>, <=, <, >=, >, ><

Table 1: Precedence of the Built-in Operators

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

Tables 2 and 3 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
monadic[1]	$+,-$	$+,-$	$+,-$	$+,-$	$+,-$	$+,-$
integer real complex	$\circ, \circ<, \circ>,$[2] $+*$	$+,-,*,/,$ $+*$	$*, *<, *>$	$*$	$*, *<, *>$	$*$
interval cinterval	$+,-,*,/,$ $+*$	$+,-,*,/,$ $+*, **$	$*$	$*$	$*$	$*$
rvector cvector	$*, *<, *>,$ $/, /<, />$	$*, /$	$\circ, \circ<, \circ>,$[3] $+*$	$+,-,*,$[4] $+*$		
ivector civector	$*, /$	$*, /$	$+,-,*,$[4] $+*$	$+,-,*,$[4] $+*, **$		
rmatrix cmatrix	$*, *<, *>,$ $/, /<, />$	$*, /$	$*, *<, *>$	$*$	$\circ, \circ<, \circ>,$[3] $+*$	$+,-,*,$[4] $+*$
imatrix cimatrix	$*, /$	$*, /$	$*$	$*$	$+,-,*,$[4] $+*$	$+,-,*,$[4] $+*, **$

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

[2] $\circ \in \{+,-,*,/\}$

[3] $\circ \in \{+,-,*\}$, where $*$ denotes the scalar or matrix product.

[4] $*$ denotes the scalar or matrix product.

$+*$: Interval hull

$**$: Interval intersection

Table 2: Predefined Arithmetical Operators

Compared with standard PASCAL, PASCAL–XSC provides an extended set of mathematical standard functions (see table 4). 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.

left operand \ right 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.

$\vee \in \{=, <>, <, <=, >, >=\}$

>< : Test on disjointedness for intervals

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

Table 3: Predefined Relational Operators

	Function	Generic Name	Argument Type
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 4: Mathematical Standard Functions ($*$ includes 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 userdefined operators, there are two ways to implement the addition of two intervals, the latter being declared by

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

One can use a procedure declaration

```
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 statement

$z := a + b + c + d$	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 description of the mathematical formulas 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 a multiple addition of intervals is described in the traditional mathematical notation. Besides the possibility of overloading operator symbols, you are allowed to use named operators. 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 $\binom{n}{k}$ can 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 arbitrary many times
- implementing recursively defined operators

The identification of the suitable operator depends on both the number and the type of the operands according to the following *weighting*-rule:

> *If the actual list of parameters matches the formal list of parameters of two different operators, then the one which is chosen has the first "better matching" parameter. "Better matching" means that the types of the operands must be consistent and not only conforming.*

Example:

```
operator +* (a: integer; b: real) irres: real;
  ⋮
operator +* (a: real; b: integer) rires: real;
  ⋮
var  x     :  integer;
     y, z  :  real;
     ⋮
  z := x +* y;  ⟹  1. operator
  z := y +* x;  ⟹  2. operator
  z := x +* x;  ⟹  1. operator
  z := y +* y;  ⟹  impossible !
```

Also, PASCAL–XSC offers the possibility to overload the assignment operator :=. Due to this, the mathematical notation may also be used 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 function symbol like *isin(...)* must be used, because the redefining of the standard function name *sin* is not allowed in standard PASCAL.

By 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 numbers of type *real*, but also for intervals, complex numbers, and other mathematical spaces. To distinguish between overloaded functions or procedures with the same name, the number, type, and weighting of their arguments are 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 new declared input/output procedure must be a **var**-parameter of file type and the second parameter represents the quantity that is to be input or output. All following 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
```

Calling an overloaded input/output procedure the file parameter may be omitted corresponding to a call with the standard files *input* or *output*. The format parameters must be introduced and seperated by colons. Moreover, several input or output statements can be combined to a single statement 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, several other programs may use the components of a module without their being copied into the source code and recompiled.

For this purpose, a module concept has been introduced in 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

Three new keywords have been added to the language:

module	:	starts a new module
global	:	indicates items to be passed to the outside
use	:	indicates imported modules

A module is introduced by the keyword **module** followed by a name and a semicolon. The body is built up quite similarly 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 equality sign in type declarations.

Thus it is possible to declare anonymous types as well as non-anonymous types. The structure of an anonymous type is not known outside the declaration module and can only be influenced by subroutine calls. If, for example, the internal structure as well as the name of a type is to be made global, then the word symbol **global** must be repeated after the equality sign. By means of the declaration

global type complex = **global record** re, im : real **end;**

the type *complex* and its internal structure as a record with components *re* and *im* is made global.
An anonymous type *complex* could be declared by

global type complex = **record** re, im: real **end;**

The user who has imported a module with this anonymous definition cannot refer to the record components, because the structure of the type is hidden inside the module.

A module is built up according to the following pattern:

> **module** m1;
> **use** < other modules >;
>   < global and local declarations >
> **begin**
>   < initialization of the module >
> **end.**

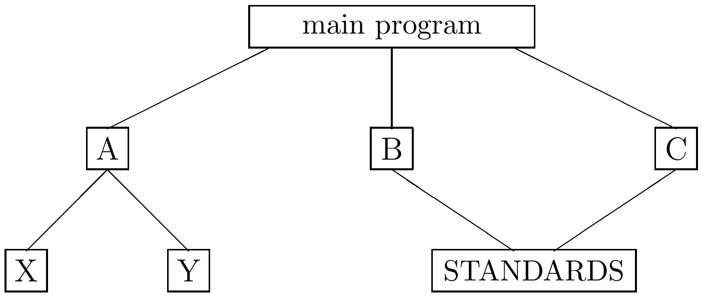
For importing modules with **use** or **use global** the following transitivity rules hold

M1 **use** M2 and M2 **use global** M3 $\Rightarrow$ M1 **use** M3.

but

M1 **use** M2 and M2 **use** M3 $\not\Rightarrow$ M1 **use** 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. There are two possibilities to make them visible in the main program, too:

1. to write

   use X, Y, STANDARDS

   in the main program
2. to write

   use global X, Y

   in module A and

   use global STANDARDS

   in module B or C.

2.5 Dynamic Arrays

In standard PASCAL there is no way to declare dynamic types or variables. For instance, program packages with vector and matrix operations can be implemented with only fixed (maximum) dimension. For this reason, only a part of the allocated memory is used if the user wants to solve problems with lower dimension only. The concept of dynamic arrays removes this limitation. In particular, the new concept can be described by the following characteristics:

- Dynamics insides of 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. There is no question of this use being fully dynamic.

In PASCAL–XSC, dynamic and static arrays can be used in the same manner. At the moment, dynamic arrays may not be components of other data structures. The syntactical meaning of this is that the word symbol **dynamic** may only be used directly following the equality sign in a type definition or directly following the colon in a variable declaration. For instance, dynamic arrays may not be record components.
A two-dimensional array type can be declared in the following manner:

```
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 strictly a 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 adaptation functions *polynomial(...)* and *vector(...)* are defined implicitly):

```
type vector = dynamic array[*] of real;
type polynomial = dynamic array[*] of real;
operator + (a,b: vector) res: vector[lbound(a)..ubound(a)];
   ⋮
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; }
```

Access to the lower and upper index limits is made possible by the new standard functions *lbound(...)* and *ubound(...)*, which are available with an optional argument for the index field of the designated dynamic variable. Employing these functions, the operator mentioned above can be written as

```
operator + (a,b: vector) res: vector[lbound(a)..ubound(a)];
var i : integer;
begin
  for i := lbound(a) to ubound(a) do
    res[i] := a[i] + b[lbound(b) + i – lbound(a)]
end;
```

Introduction of dynamic types requires an extension of the compatibility prerequisites. As in standard PASCAL, two array types are not compatible unless they are of the same type. Consequently, a dynamic array type is not compatible with a static type. In PASCAL–XSC value assignments are always possible in the cases listed in Table 5.

Type of Left Side	Type of Right Side	Assignment Permitted
anonymous dynamic	arbitrary *array* type	if structurally equivalent
known dynamic	known dynamic	if types are the same
anonymous static	arbitrary *array* type	if structurally equivalent
known static	known static	if types are the same

Table 5: Assignment Compatibilities

In the remaining cases, an assignment is possible only for equivalent qualification of the right side (see [14] or [15] for details).

In addition to access to each component variable, PASCAL–XSC offers the possibility of access to entire subarrays. If a component variable contains an $*$ instead of an index expression, it refers to the subarray with the entire index range in the corresponding dimension, e. g. via *m[*, j]* the *j*-th column of a two-dimensional array *m* is accessed. 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 adaptation function, since both the left-hand and the right-hand 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 adaptation function *vector(...)*.

A PASCAL–XSC program which uses dynamic arrays should be built up according to the following scheme:

```
program dynprog (input,output);
type
  vector = dynamic array[∗] of real;
  < different dynamic declarations >
var n : integer;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
procedure main (dim: integer);
var a,b,c : vector[1..dim];
  ⋮
begin
  < I/O depending on the value of dim >
    ⋮
  c := a + b;
    ⋮
end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
begin {main program}
  read(n);
  main(n);
end. {main program}
```

It is necessary to frame only the original main program by a procedure (here: *main*), which is refered to with the dimension of the dynamic arrays as a transfer parameter.

2.6 Accurate Expressions

The implementation of inclusion algorithms with automatic result verification or validation (see [12],[18],[22],[27]) makes extensive use of the accurate evaluation of dot products with the property (see [19])

$$\text{(RG)}\quad a \bigodot b := \bigcirc \sum_{i=1}^{n} a_i \cdot b_i,\ \ \bigcirc \in \{\square, \triangle, \triangledown\},\ \ n \in I\!N.$$

To evaluate this kind of expression the new datatype *dotprecision* was introduced. This datatype accomodates the full floating-point range with double exponents (see [19],[18]). 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 and is evaluated without any rounding error. The following standard operations are available for *dotprecision*:

- conversion of *real* and *integer* values to *dotprecision* (#)
- rounding of *dotprecision* values to *real*; in particular: downwardly directed rounding (#<), upwardly directed rounding (#>), and rounding to the nearest (#*)
- rounding of a *dotprecision* expression to the smallest enclosing interval (##)
- addition of a *real* number or the product of two *real* numbers to a variable of type *dotprecision*
- addition of a dot product to a variable of type *dotprecision*
- addition and subtraction of *dotprecision* numbers
- monadic minus of a *dotprecision* number
- the standard function *sign* returns −1, 0, or +1, depending on the sign of the *dotprecision* number

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 6 shows the possible rounding modes with respect to the dot product expression form (see the appendix on page 27 for details).

Symbol	Expression Form	Rounding Mode
#*	scalar, vector or matrix	nearest
#<	scalar, vector or matrix	downwards
#>	scalar, vector or matrix	upwards
##	scalar, vector or matrix	smallest enclosing interval
#	scalar only	exact, no rounding

Table 6: 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

$$\sum_{k=1}^{n} A(i,k) \cdot B(k,j)$$

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 defect expressions. Let, for instance, in case of a linear system $Ax = b$, $A \in I\!R^{n\times n}$, $x, b \in I\!R^n$, be $Ay \approx b$. Then an enclosure of the defect is given by $\Diamond(b - Ay)$ and is realized in PASCAL–XSC via

##(b − A*y);

with only one interval rounding operation per component. To get verified inclusions of linear systems of equations it is necessary to evaluate the defect expression

$$\Diamond(E - RA)$$

were $R \approx A^{-1}$ 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 a part of the module for real matrix/vector arithmetic and generates an identity matrix of appropriate dimension according to the shape of A (see section 2.8).

2.7 The String Concept

The tools provided for handling strings in standard PASCAL do not enable convenient text processing. For this reason, a string concept was integrated into the language definition of PASCAL–XSC which allows a comfortable handling of textual information and even symbolic computation. With this new data type *string*, the user can work with strings of up to 255 characters. Provided a string doesn't exceed a certain range, the user can specify the length in the *string* declaration part. Thus a string *s* declared by

> **var** s: string[40];

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

- concatenation

  operator + (a,b: string) conc: string;

- actual length

  function length(s: string): integer;

- conversion *string* → *real*

  function rval(s: string): real;

- conversion *string* → *integer*

  function ival(s: string): integer;

- conversion *real* → *string*

  function image(r: real; width,fracs,round: integer): string;

- conversion *integer* → *string*

  function image(i,len: integer): string;

- extraction of substrings

  function substring(s: string; i,j: integer): string;

- position of first appearance

  function pos(sub,s: string): integer;

- relational operators <=, <, >=, >, <>, =, and **in**

2.8 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 7 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 7: Predefined Arithmetical 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 respectively, 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 are determined via *diam* and *mid*, *blow* yields an interval inflation and *transp* is used to get the transposed 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.

2.9 Problem-Solving Routines

PASCAL–XSC routines for solving common numerical problems are supplied by means of an additional module library. The applied methods compute a highly accurate inclusion of the true solution of the problem and, at the same time, prove existence and uniqueness of the solution in the given interval. The advantages of these new routines are listed in the following:

- The solution is computed with maximum or highly, but always controlled accuracy, even in many ill-conditioned cases.
- The correctness of the result is automatically verified, i. e. an inclusion set is computed which guarantees existence and uniqueness of the exact solution within.
- In case no solution exists or the problem is extremely ill-conditioned, an error message is indicated.
- Since the user has almost no chance to make an error or to misinterpret something, these routines may also be applied by non-specialists.

Particularly, 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
- noninear systems of equations
- numerical quadrature
- integral equations
- automatic differentiation

3 The Implementation of PASCAL–XSC

Since 1976, a PASCAL extension for scientific computation has been in the process of being defined and developed at the Institute for Applied Mathematics at the University of Karlsruhe. The PASCAL-SC compiler has been implemented on several computers (Z80, 8088, and 68000 processors) under various operating systems. This compiler was already in the marketplace for the IBM PC/AT and the ATARI-ST (see [16], [17]).

The new PASCAL–XSC compiler is now available for personal computers, workstations, mainframes, and supercomputers by means of an implementation in C. Via a PASCAL–XSC-to-C precompiler and a runtime system implemented in C the language PASCAL–XSC may be used, among other systems, on all UNIX systems in an almost identical way. Thus, the user has the possibility to develop his programs for example on a personal computer and afterwards get them running on a mainframe via the same compiler.

A complete description of the language PASCAL–XSC and the arithmetic modules as well as a collection of sample programs is given in [14] and [15].

4 PASCAL–XSC Sample Program

In the following, a complete PASCAL–XSC program is listed, which demonstrates the use of some of the arithmetic modules. Using the module LIN_SOLV, the solution of a system of linear equations is enclosed in an interval vector by succecsive 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 of procedure *linear_system_solver* defined in module LIN_SOLV. This procedure may be called with arbitrary dimension of the used arrays.

For detailed information on iteration methods with automatic result verification see [12], [18], [22], or [26], for example.

Main Program

```
program lin_sys (input,output);

{ Program for verified solution of a linear system of equations. The    }
{ matrix A and the right-hand side b of the system are to be read in. }
{ The program delivers either a verified solution or a corresponding    }
{ failure message.                                                      }

use                            { lin_solv : linear system solver              }
  lin_solv, mv_ari, mvi_ari;   { 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}
```

Module LIN_SOLV

```
module lin_solv;

{ Verified solution of the linear system of equations Ax = b. }

use                     { i_ari   : interval arithmetic                }
  i_ari, mv_ari, mvi_ari; { 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..ubound(a)];

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

var
  i  : integer;
  x : interval;

begin
  for i:= 1 to ubound(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..ubound(A),1..ubound(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..ubound(A),1..ubound(A)];

begin
  n := ubound(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 }
  { a probably singular matrix A.                    }

  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 inclusion vector for the solution of the }
{ linear system of equations. If an inclusion 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..ubound(A)];
  R    : rmatrix[1..ubound(A),1..ubound(A)];
  C    : imatrix[1..ubound(A),1..ubound(A)];
begin
  R := approximate_inverse(A);

  { R*b is an approximate solution of the linear system and z is an inclusion }
  { of this vector. However, it does not usually include the true solution.    }
  z := ##(R*b);

  { An inclusion 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 inflated epsilon; { To obtain a true inclusion 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}
```

Appendix

Review of 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*

Review of 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*

References

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

[2] Bleher, J. H., Rump, S. M., Kulisch, U., Metzger, M., Ullrich, Ch., and Walter, W.: *FORTRAN-SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH.* Computing **39**, 93 - 110, 1987.

[3] Bohlender, G., Grüner, K., Kaucher, E., Klatte, R., Krämer, W., Kulisch, U., Rump, S., Ullrich, Ch., Wolff von Gudenberg, J., and Miranker, W.: *PASCAL-SC: A PASCAL for Contemporary Scientific Computation.* Research Report RC 9009, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981.

[4] Bohlender, G., Grüner, K., Kaucher, E., Klatte, R., Kulisch, U., Neaga, M., Ullrich, Ch., and Wolff von Gudenberg, J.: *PASCAL-SC Language Definition.* Internal Report of the Institute for Applied Mathematics, University of Karlsruhe, 1985.

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

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

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

[8] Däßler, K. und Sommer, M.: *PASCAL, Einführung in die Sprache.* Norm Entwurf DIN 66256, Erläuterungen. Springer, Berlin, 1983.

[9] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In: [27], 1990.

[10] IBM *High-Accuracy Arithmetic Subroutine Library* (ACRITH). General Information Manual, GC 33-6163-02, 3rd Edition, 1986.

[11] IBM *High-Accuracy Arithmetic Subroutine Library* (ACRITH). Program Description and User's Guide, SC 33-6164-02, 3rd Edition, 1986.

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

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

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

[15] Klatte, R., Kulisch, U., Neaga, M., Ratz, D. und Ullrich, Ch.: *PASCAL–XSC Language Reference with Examples.* To be published by Springer, Heidelberg, 1991/92.

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

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

[18] Kulisch, U. (Hrsg.): *Wissenschaftliches Rechnen mit Ergebnisverifikation – Eine Einführung.* Akademie Verlag, Ost-Berlin, Vieweg, Wiesbaden, 1989.

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

[20] Kulisch, U. and Miranker, W. L.: *The Arithmetic of the Digital Computer: A New Approach.* SIAM Review, Vol. 28, No. 1, 1986.

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

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

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

[24] Neaga, M.: *PASCAL-SC – Eine PASCAL-Erweiterung für wissenschaftliches Rechnen.* In: [18], 1989.

[25] Ratz, D.: *The Effects of the Arithmetic of Vector Computers on Basic Numerical Methods.* In: [27], 1990.

[26] Rump, S. M.: *Solving Algebraic Problems with High Accuracy.* In: [21], 1983.

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

[28] Wolff von Gudenberg, J.: *Einbettung allgemeiner Rechnerarithmetik in PASCAL mittels eines Operatorkonzeptes und Implementierung der Standardfunktionen mit optimaler Genauigkeit.* Dissertation, Universität Karlsruhe, 1980.