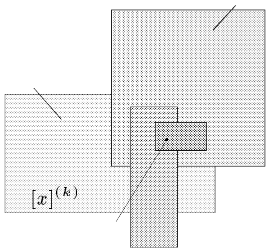
Institut für **A**ngewandte **M**athematik
Universität **K**arlsruhe (TH)

PASCAL–XSC
BCD-Version 1.0

Benutzerhandbuch
für das
dezimale Laufzeitsystem

Frithjof Blomquist



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

PASCAL–XSC
BCD-Version 1.0

Benutzerhandbuch für das dezimale Laufzeitsystem

Frithjof Blomquist

Jeder Rechner liefert wegen unvermeidbarer Rundungsfehler z.B. für die Addition zweier Gleitkommazahlen eine i.a. fehlerbehaftete Summe, so daß man durch **naive** Gleitkommarechnungen Maschinenergebnisse erhält, über deren Fehler **gesicherte** Aussagen bei umfangreichen Algorithmen mit möglicherweise Hunderttausenden von Gleitkommaoperationen nicht möglich sind.

Zur Behebung dieser aus mathematischer Sicht sehr unbefriedigenden Situation stellt die **PASCAL-XSC** Umgebung Werkzeuge zur Verfügung, mit denen für viele mathematische Standardprobleme eine sehr genaue und vor allem **garantierte** Einschließung der jeweils exakten Lösung berechnet werden kann. Dazu müssen lediglich entsprechende Routinen, etwa zur Lösung linearer Gleichungssysteme, aus Hilfsmodulen aufgerufen werden, wobei ein Hintergrundwissen bzgl. der benutzten, selbstverifizierenden Algorithmen nicht erforderlich ist!

In diesem Handbuch findet der Anwender zur Entwicklung eigener, selbstverifizierender Algorithmen eine ausführliche Beschreibung der verfügbaren Zahlenformate mit 13 bzw. 21 Dezimalziffern und 47 **hochgenaue** Standardfunktionen für Punkt- und Intervallargumente. Mit einer Vielzahl von Funktionen, Prozeduren und Operatoren für reelle und komplexe Operanden läßt sich die **PASCAL-XSC** Umgebung mit dem Modulkonzept beliebig erweitern.

Die vorliegende **BCD**–Version ermöglicht das **rundungsfehlerfreie** Abspeichern dezimaler Eingabedaten, so daß die bei der **IEEE**–Version oft so lästigen Konversionsfehler entfallen!

Vorwort

Zweifellos hat die Computer–Technik in den letzten Jahrzehnten gewaltige Fortschritte gemacht; so kann man ohne Übertreibung sagen, daß sich die Prozessorleistungen geradezu explosionsartig entwickelt haben. Betrachtet man insbesondere noch die zukünftigen Möglichkeiten der Parallelverarbeitung mit mehreren Prozessoren, so ist ein Ende dieser Leistungssteigerungen heute keinesfalls absehbar.

Nun könnte man vermuten, daß der Computer, ähnlich etwa wie in der Multi-Media-Technik, auch für den Bereich der Mathematik die gleichen gewaltigen Fortschritte gebracht hätte. Aus der Sicht der Mathematiker ist jedoch die Prozessorentwicklung, von der Geschwindigkeitssteigerung einmal abgesehen, in ihrem Anfangsstadium steckengeblieben. So werden bis heute lediglich die vier Grundoperationen in maximaler Genauigkeit und eine recht kleine Anzahl von Standardfunktionen bereitgestellt, deren Fehlerschranken, aus welchen Gründen auch immer, nicht zu erfahren sind. Mit diesen minimalen Prozessormöglichkeiten wurden bisher numerische Algorithmen entwickelt, die wegen unvermeidbarer Rundungsfehler bei den Grundoperationen für die mathematisch exakten Ergebnisse lediglich Näherungen geliefert haben. Für den Mathematiker sind solche Näherungen jedoch völlig unbrauchbar, wenn deren Güte nicht bekannt ist. Man benötigt daher entweder die Berechnung einer garantierten Fehlerschranke, oder man muß das anstehende numerische Problem mit einem selbstverifizierenden Algorithmus lösen, der automatisch eine möglichst enge Intervalleinschließung der exakten Lösung liefert. Beide Aufgaben lassen sich mit den **XSC**–Sprachen wie **PASCAL-XSC** , **FORTRAN-XSC** oder **C-XSC** realisieren, die am Institut für Angewandte Mathematik der Universität Karlsruhe unter der Leitung von Professor U. Kulisch entwickelt wurden.

Mit diesen Werkzeugen ist man heute in der Lage, nur aus numerischen Gleitkommaergebnissen wirkliche mathematische Schlüsse zu ziehen. Hat man z.B. für $x \in [a,b]$ eine Einschließung $Y = [y_1, y_2]$ des Wertebereichs der ersten Ableitung f' einer Funktion $f : [a,b] \longrightarrow \mathbb{R}$ berechnet, und gilt für das Gleitkommaergebnis y_1 die Beziehung $y_1 > 0$, so ist mit dem Computer der Beweis geführt worden, daß die Funktion f in $[a,b]$ streng monoton wächst. Beachten Sie bitte, daß sich diese Arbeitsweise von den sonst üblichen numerischen Methoden ganz wesentlich unterscheidet. Bisher hat man nämlich meist darauf vertraut, daß das Ergebnis einer Computerrechnung wenigstens auf den ersten Ziffern korrekt sein wird, wenn man instabile Teile des Algorithmus z.B. in doppelter Genauigkeit auswertet.

Mit diesen Methoden lassen sich natürlich keine mathematischen Beweise führen, zumal man auch bei doppeltgenauen Rechnungen immer wieder Problemstellungen findet, die nicht einmal annähernd korrekt gelöst werden können.

Die schon genannten **XSC**-Sprachen liefern zwar alle wesentlichen Hilfsmittel, die für eine mathematisch zuverlässige Numerik erforderlich sind, diese Hilfsmittel müssen jedoch alle softwaremäßig simuliert werden, da sie bis heute nicht direkt durch einen Prozessor realisiert worden sind. Durch die notwendigen Software–Simulationen besitzen die **XSC**-Systeme natürlich im Vergleich zur herkömmlichen Numeriksoftware einen großen Laufzeitnachteil. Aus Sicht des Mathematikers wäre es daher wünschenswert, zunächst wenigstens die vier Grundoperationen der Intervallrechnung direkt durch den Prozessor bereitzustellen, da die Intervallarithmetik insbesondere bei den selbstverifizierenden Algorithmen ein unentbehrliches Hilfsmittel ist.

In dem folgenden Beispiel wird aus mathematischer Sicht eine weitere ganz wesentliche Schwäche der heutigen Prozessorgeneration demonstriert. Mit einem Taschenrechner soll dazu in zehnstelliger Dezimalarithmetik die Differenz $a \cdot b - c$ berechnet werden. Mit den Werten $a = b = 1234567891$ und $c = 1.524157877 \cdot 10^{18}$ erhält man an Stelle des exakten Wertes $a \cdot b - c = 4.88187881 \cdot 10^8$ das völlig falsche Taschenrechnerergebnis 0, d.h. schon nach zwei Grundoperationen erhält man noch nicht einmal die erste Ziffer des korrekten Ergebnisses. Der Grund liegt darin, daß das exakte Produkt $a \cdot b = 1524157877488187881$ vor der Subtraktion von c bei diesem Beispiel genau auf den Wert von c abgerundet wird, d.h. der Mantissenanteil, der das exakte Ergebnis repräsentiert, geht durch den Rundungsfehler verloren. Ganz entsprechende Beispiele lassen sich auch für PC's oder Workstations angeben, wobei jedoch die Zahlenangaben in dem für uns unlesbaren Binärsystem erfolgen müßten. Für den Mathematiker ist diese Situation natürlich eine einzige Katastrophe, da die im obigen Beispiel angesprochene Skalarproduktberechnung z.B. beim Lösen von linearen Gleichungssystemen ein zentraler Punkt ist, und auch Rechnungen im doppeltlangen Zahlenformat die kontrollierte Auswertung eines Skalarproduktes bei höheren Dimensionen nicht gewährleisten. Dies wird erst durch die **XSC**-Sprachen ermöglicht, die softwaremäßig einen Festkommaspeicher (Akkumulator) bereitstellen, in den man alle möglichen Teilprodukte in beliebiger Anzahl **rundungsfehlerfrei** addieren kann. Da die Simulation dieses Akkumulators in Software recht zeitaufwendig ist, wurde unter Federführung des Instituts für Angewandte Mathematik der Universität Karlsruhe ein Coprozessor entwickelt, mit dem Skalarprodukte beliebiger Dimension in Hardware exakt berechnet werden können. Dabei zeigte sich, daß die Laufzeit sogar kürzer ist als bei der Auswertung mit Hilfe der Grundoperationen $*, +$, die zudem noch keine kontrollierten Ergebnisse liefern. Es wäre daher schon ein großer Fortschritt, wenn man auf einem Mathematik–Prozessor neben einer Intervallarithmetik auch ein oder zwei (für Parallelverarbeitung) Festkommaspeicher realisieren würde!

Aber selbst ein solcher Prozessor hätte für die Anforderungen der Mathematiker immer noch einen schwerwiegenden Nachteil, da die Arithmetiken aller heutigen Prozessoren in PC's und Workstations auf dem binären Zahlensystem basieren, während alle Mathematiker ihre Probleme im dezimalen Zahlensystem formulieren, ganz einfach, weil man die Größe einer Zahl im Zehnersystem am einfachsten erfassen kann. Soll nun z.B. mit $a = 9.000000000000001$ und $b = 9$ die Differenz $a - b$ berechnet werden, so müssen a, b nach der Tastatureingabe jeweils zur nächsten Binärzahl gerundet werden, was mit der Dezimalzahl a nicht fehlerfrei möglich ist. Die anschließende exakte Subtraktion liefert dann das völlig unbrauchbare Ergebnis $1.776.. \cdot 10^{-15}$, das nur durch den Konversionsfehler bei der Rundung von a ins Binärsystem begründet ist. Im Gegensatz dazu liefert nun jeder Taschenrechner mit einer dezimalen Arithmetik bei 16stelliger Mantisse die exakte Differenz $1 \cdot 10^{-15}$, und vermutlich könnte man nicht einen einzigen Taschenrechner verkaufen, der für obige Differenz den falschen Wert $1.776.. \cdot 10^{-15}$ berechnet. Auf den Seiten 4 ... findet man weitere Beispiele, die den oft so lästigen Einfluß der Konversionsfehler eindrucksvoll demonstrieren. Der Leser sollte diese Beispiele sehr sorgfältig studieren, da die Erfahrung zeigt, daß selbst versierte Anwender numerischer Software die Folgen dieser Konversionsfehler nicht immer richtig einschätzen!

Es war somit ein naheliegender Schritt, eine **PASCAL-XSC** BCD-(**B**inary **C**oded **D**ecimal)-Version zu entwickeln, die alle Sprachelemente der Binärversion umfaßt. Dem Anwender stehen zwei Datenformate (real, Longreal) mit 13 bzw. 21 Dezimalziffern zur Verfügung, wobei das vollständige Akkumulatorkonzept wie bei der Binärversion auch für das real–Format existiert. Im Longreal–Format sind alle Additionen in den Akkumulator möglich, während die Multiplikation bzgl. des Exponentenbereichs auf das real–Format eingeschränkt ist. Mit einer Vielzahl von Hilfsroutinen läßt sich das Akkumulatorkonzept vollständig auf das Longreal–Format übertragen. Damit wird z.B. auch die komplexe Intervallrechnung mit 21 Dezimalstellen ermöglicht. Für beide Datenformate existieren jeweils 47 hochgenaue Standardfunktionen mit Angabe der jeweiligen relativen Fehlerschranke. Dem Anwender steht ein weiteres Datenformat (rrG) mit 26 Dezimalstellen zur Verfügung, dessen Exponentenbereich nur durch die integer–Zahlen begrenzt ist. In diesem Format können daher Zwischenrechnungen praktisch ohne Über- oder Unterlauf durchgeführt werden, und zu den 47 schon existierenden Standardfunktionen lassen sich weitere Funktionen für reelle oder komplexe Punkt- bzw. Intervallargumente entwickeln. Durch Rundung der Funktionswerte ins Longreal–Format erhält man dann zusätzliche **hochgenaue** Anwenderfunktionen, d.h. das BCD–System läßt sich beliebig erweitern. Zusammen mit den vielen mitgelieferten Hilfsroutinen dieser Version besitzt der Anwender ein sehr mächtiges Werkzeug, mit dem sich aus garantierten numerischen Ergebnissen wirklich mathematische Aussagen gewinnen lassen. Wenn später einmal ein Prozessor mit einer dezimalen Intervallarithmetik und einigen Festkommaspeichern zur Verfügung steht, dann wird man die jetzt bei aufwendigen Anwendungen unbefriedigenden Laufzeiten noch wesentlich verbessern können.

Abschließend sei noch erwähnt, daß auch mir bekannt ist, daß eine Programmiersprache niemals abgeschlossen und vollkommen und stets noch verbesserungsfähig ist. Wohlwollende kritische Bemerkungen und Verbesserungsvorschläge werden daher gerne entgegengenommen.

Karlsruhe, im Juli 1997 — Dr. Frithjof Blomquist

Zur Entstehung dieses Buches

In einer ersten Version wurden die Standardfunktionen im 21-stelligen Longreal-Format mit einer relativen Fehlerschranke von ca. 10^{-18} entwickelt. Für Punktargumente ergaben sich daher im Vergleich zu den exakten Funktionswerten Abweichungen bei den letzten zwei oder drei Mantissenziffern, und entsprechend waren die Funktionswerteinschließungen aufgebläht. Es schien mir dann jedoch ratsam zu sein, dem Anwender eines Mathematik-Paketes, das verifizierendes Rechnen ermöglicht, Standardfunktionen anzubieten, deren Maschinenergebnisse sich vom optimal gerundeten Funktionswert auf der letzten Mantissenstelle höchstens um **eine Einheit** unterscheidet. Zur Implementierung solcher **hochgenauen** Standardfunktionen im Longreal-Bereich wurden dann in einer zweiten Version die entsprechenden Algorithmen in einem doppeltlangen real-Format mit 26 BCD-Stellen realisiert. Insbesondere für die Intervallfunktionen waren zusätzliche Hilfsfunktionen und Programme notwendig, so daß die entsprechende Sammlung von Notizen und Anmerkungen immer unübersichtlicher wurde. Auch die Vielzahl der notwendigen Operatoren für die verschiedenen Datentypen real,Longreal, interval,Linterval, ... veranlaßte mich schließlich, die vorliegende Blattsammlung mit dem Textsystem LaTeX in Buchform zu bringen. Dabei war es nicht mein Ziel, eine neue Sprachbeschreibung für PASCAL–XSC zu entwickeln, da dies bereits durch die Autoren R. Klatte, U. Kulisch, M. Neaga, D. Ratz und Ch. Ullrich mit dem Buch: *PASCAL–XSC, Sprachbeschreibung mit Beispielen* realisiert wurde. Im vorliegenden Buch werden vielmehr Abweichungen von der Binärversion und Besonderheiten der BCD–Version beschrieben und mit vielen Beispielen belegt. Dadurch erscheint die Gliederung nicht immer zwingend systematisch; ein umfangreiches Stichwortverzeichnis soll dem Leser deshalb eine zusätzliche Orientierungsstütze sein. Mit Hilfe dieses Buches, zusammen mit der oben genannten Sprachbeschreibung, sollte der Leser in der Lage sein, verifizierende Algorithmen selbst zu realisieren und damit den Rechner als ein wirkliches Hilfsmittel des Mathematikers weiterzuentwickeln.

Ich möchte an dieser Stelle meinen Dank an U. Kulisch, W. Krämer und H. Berlejung aussprechen, die mit vielen Anregungen und Hilfen am Zustandekommen dieses Buches beteiligt waren.

Inhaltsverzeichnis

Kapitel 1

PASCAL–XSC im Internet

Da ein PASCAL–XSC-Compiler nach **C** übersetzt, kann ein PASCAL–XSC-System praktisch auf jedem Rechner installiert werden, der über einen C–Compiler verfügt.

Frei im Internet abrufbar sind derzeit PASCAL–XSC Compiler für DOS, OS/2 und LINUX. Die Software und weitere Informationen finden Sie auf der Homepage der Firma

Numerik Software GmbH
P.O. BOX 2232
D–76492 Baden–Baden
Germany
Tel.: +49 7221 949217
Fax: +49 7221 949218
email: 100145.2667@compuserve.com
www: http://come.to/xsc–software

Auszug aus der Homepage der Firma Numerik Software GmbH

The PASCAL-XSC package contains:

The compiler from PASCAL-XSC to C, modules for interval arithmetic, complex arithmetic, complex interval arithmetic and the corresponding matrix/vector arithmetic modules and the PASCAL-XSC runtime library for the corresponding C compiler. Try to test PASCAL-XSC and download a DOS or OS/2 Public Domain version of the PASCAL-XSC compiler system running under EMX GNU-C emx0.9b or a Linux version running under GNU-C. You can also download a Toolbox for PASCAL-XSC or C-XSC including some numerical sample programs which are described in the book Numerical Toolbox for Verified Computing I resp. C++ Toolbox for Verified Computing.

The C-XSC software package contains:

Object modules and header files for interval arithmetic, complex arithmetic, complex interval arithmetic and the corresponding matrix/vector arithmetic modules, a staggered multiple precision arithmetic, application examples, the C-XSC runtime library for the corresponding C++ compiler and the sources from the Toolbox book (see CXSCTOOL95). Try to test the C-XSC class library and download a DOS Public Domain version of the C-XSC compiler system running under Borland C++ 4.0.

The corresponding language descriptions are not part of the software package and must be ordered separately. Schools and universities can have a discount of 40% for Unix based system packages. This rebate can be given only for educational use of the system. Beneficiaries are students, teachers and professors at schools, universities and comparable institutions. The prices are valid for one CPU of the corresponding target computer system. Licences can be granted for multiprocessor machines as well as for network systems. Campus licences are available on request. The systems can be adapted to further target machines on request. The license includes free updates for the first license year. We only charge for shipment costs.

Shipment for companies and public institutions inside Europe on account, outside Europe payment in advance, per C.O.D. or with credit cards.

Forwarding Expense: inside Europe: 30,00 DM, outside Europe: 50,00 DM.

Accepted credit cards: American Express, Master Card and Visa. For EC countries the VAT ID-number is necessary to allow VAT free invoices.

Kapitel 2

Vorteile eines BCD–Formats

2.1 Binär–Format $\longleftrightarrow$ BCD–Format

Die Vorteile eines BCD–Zahlenformats lassen sich am einfachsten durch die Nachteile z.B. eines internen Binärsystems beschreiben. Die folgende Abbildung 2.1 zeigt dazu das typische Schema der numerischen Datenverarbeitung mit einem binären Zahlenformat. Die ergebnisverifizierenden Algorithmen beziehen sich dabei auf die Binärversion von PASCAL–XSC:

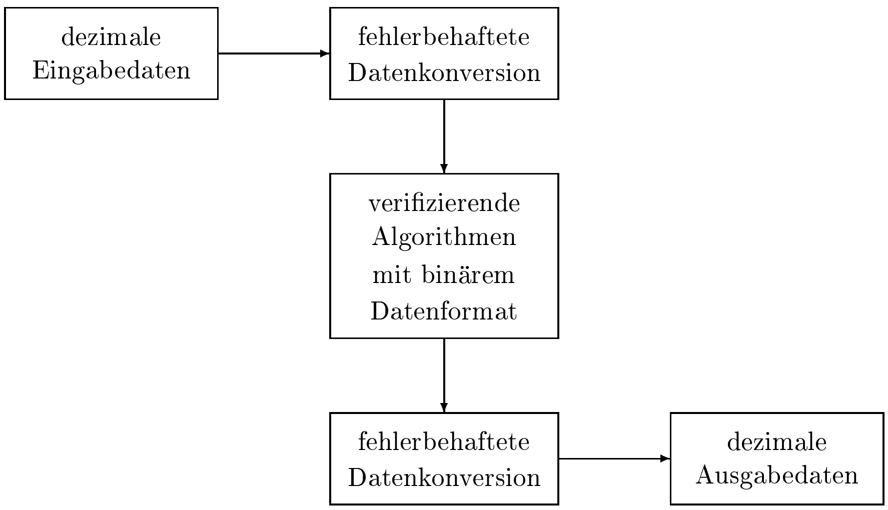


Abbildung 2.1: Datenverarbeitung mit binärem Zahlenformat

Daß die heutigen elektronischen Rechenanlagen intern ein duales Datenformat benutzen, hat rein historische Gründe, da jeder Rechner intern nur über **zwei** Grundzustände verfügt, die z.B. mit 0 oder 1 bezeichnet werden. Es war daher sehr naheliegend, interne Zahlenformate mit der Basis 2^n zu benutzen (Binär-, Oktal-, Hexadezimal–Systeme). Man brauchte lediglich Konversionsroutinen, um die Eingabedaten aus dem uns gewohnten Zehnersystem z.B. ins Dual–System zu übertragen. Nach dem Programmablauf ist dann noch die Umwandlung der Ergebnisdaten in das von uns lesbare Zehnersystem notwendig. Zur Berechnung garantierter Ergebniseinschließungen muß man dabei in PASCAL–XSC natürlich entsprechend aufrunden, wenn z.B. eine Oberschranke ausgegeben werden soll!

Die Vorteile der internen Zahlensysteme mit der Basis 2^n sind einmal die effiziente Speichernutzung und die kurzen Laufzeiten der Grundoperationen $+, -, *, /$. Man könnte also der Auffassung sein, daß damit alle Grundvoraussetzungen erfüllt sind, einen Rechner in allen mathematischen Bereichen erfolgreich einzusetzen!

Leider ist aber die exakte Umwandlung der Dezimalzahlen in ein duales Zahlenformat endlicher Länge **nur in Ausnahmefällen** möglich, d.h. schon die Umwandlung von 0.1 ins Binärsystem erzeugt **zwangsläufig** einen Fehler, der zwar um so kleiner ist, je breiter die benutzte, interne Mantisse ist, der aber immer von Null verschieden sein wird!

Bei der Verarbeitung von Meßdaten, die grundsätzlich schon selbst mit Fehlern behaftet sind, haben die zusätzlichen Konversionsfehler keinen entscheidenden Einfluß auf die Ergebnisse. Die folgenden fünf mit der Binärversion von PASCAL–XSC durchgerechneten Beispiele zeigen jedoch, daß Konversionsfehler bei der Umwandlung von Dezimalzahlen ins Binär–System schon bei einfachen **mathematischen** Problemstellungen eine ganz entscheidende **negative** Rolle spielen können.

In den nachfolgenden Beispielen bedeutet $\widetilde{a}$ die zur vorgegebenen Dezimalzahl a nächstgelegene Maschinenzahl. Für $a = 0.1$ gilt dann $\widetilde{a} \neq a$, und für $a = 8$ sind $\widetilde{a}$ und a identisch. Eingekreiste Operatorsymbole bedeuten Maschinenoperationen, die i.a. mit Rundungsfehlern behaftet sind, während nicht eingekreiste Operatoren stets exakte Ergebnisse erzeugen.

1. Beispiel

$a = 9.000000000000001$; $b = 9$; exakter Wert: $a - b = 10^{-15}$;
Das interne Rechnerergebnis $\widetilde{a} \ominus \widetilde{b}$ der Binär–Version lautet jedoch ins Zehnersystem zurückgerundet: $1.776356839400250 \cdot 10^{-15}$. Die notwendige Zahlenkonversion zusammen mit der nachfolgenden Subtraktion erzeugt damit den sehr großen relativen Fehler von $0.776\ldots$! Eine entsprechende intervallmäßige Auswertung liefert daher für den exakten Wert 10^{-15} die grobe Einschließung:

$$10^{-15} \in [0;\ 1.776356839400251 \cdot 10^{-15}].$$

2. Beispiel

$x_0 = 9.424777960769380 \approx 3 \cdot \pi$;
exakter Wert: $\sin(x_0) = -2.846120698501614913474\ldots \cdot 10^{-16}$;
Das interne Rechnerergebnis $\widetilde{\sin}(\widetilde{x}_0)$ der Binär–Version lautet ins Zehnersystem zurückgerundet: $+3.673940397442059 \cdot 10^{-16}$, d.h. der

Rechner liefert noch nicht einmal das richtige Vorzeichen des gesuchten Funktionswertes ! Eine Intervallauswertung ergibt entsprechend die sehr grobe Einschließung:

$$\sin(x_0) \in [-1.408962799656045 \cdot 10^{-15};\ +3.673940397442060 \cdot 10^{-16}].$$

Diese, durch Konversionsfehler bedingten Überschätzungen, findet man immer dann, wenn eine Funktion in der unmittelbaren Umgebung einer Nullstelle auszuwerten ist !

3. Beispiel

$a = b = 1.234567890123456;\quad c = 1.234567890123457;$
$d = 1.234567890123455;$ exakter Wert: $a \cdot b - c \cdot d = 10^{-30};$
Das interne Skalarprodukt $\#*(\widetilde{a} \cdot \widetilde{b} - \widetilde{c} \cdot \widetilde{d})$ lautet ins Zehnersystem zurückgerundet: $2.741291394155932 \cdot 10^{-16}$, d.h. im Vergleich zum exakten Wert 10^{-30} erhalten wir mit dem binären Ergebnis eine Abweichung um 14 Größenordnungen, bedingt durch die anfänglichen Konversionen: $a \to \widetilde{a},\ b \to \widetilde{b}, \ldots$.
Betrachtet man die große Abweichung vom exakten Ergebnis, so erinnert der obige Einsatz des langen Akkumulators deutlich an das Schießen mit Kanonen auf Spatzen, denn in der Tat liefert die Auswertung von $\widetilde{a} \odot \widetilde{b} \ominus \widetilde{c} \odot \widetilde{d}$ den Wert $2.22\ldots \cdot 10^{-16}$, der sich vom Akkuergebnis nur unwesentlich unterscheidet!
Um eine **Einschließung** von $a \cdot b - c \cdot d$ zu erhalten, müssen zunächst die vorgegebenen Dezimalzahlen a, b, c, d durch Binärintervalle ai, bi, ci, di kleinsten Durchmessers eingeschlossen werden. Die anschließende Auswertung des Ausdrucks $\#\#(ai \cdot bi - ci \cdot di)$ liefert dann die für die Praxis viel zu grobe Einschließung:

$$a \cdot b - c \cdot d \in [-8.22\ldots \cdot 10^{-16},\ +2.74\ldots \cdot 10^{-16}]$$

Das Beispiel zeigt weiter, welche Überschätzungen etwa beim Lösen fast singulärer, linearer Gleichungssysteme zu erwarten sind, wenn die vorgegebenen dezimalen Punktkoeffizienten zunächst in entsprechende Binärintervalle kleinsten Durchmessers eingeschlossen werden müssen. Wir betrachten dazu das

4. Beispiel

Zu lösen ist das lineare Gleichungssystem: $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ mit:

$$\mathbf{A} = \begin{pmatrix} c & a \\ b & d \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

wobei die Zahlen a, b, c, d die gleichen Werte haben wie im Beispiel 3. Die exakte Lösung lautet:

$$x_1 = \frac{a}{a \cdot b - c \cdot d} = +1.234567890123456 \cdot 10^{30}$$
$$x_2 = \frac{-c}{a \cdot b - c \cdot d} = -1.234567890123457 \cdot 10^{30}$$

Versucht man nun eine Lösung des Systems mit Hilfe der Prozedur LSS aus dem Modul ILSS der Binär–Version von PASCAL–XSC, so wird das Programm mit dem Errorcode 2 abgebrochen, da die interne Matrix $\widetilde{\mathbf{A}}$ für das System singulär erscheint. Die Koeffizienten a, b, c, d müssen nämlich vor Beginn der eigentlichen Rechnung in Binärintervalle eingeschlossen werden, deren Durchmesser alle von Null verschieden sind, so daß intern nicht mit $\mathbf{A}$ sondern mit der Intervallmatrix $\widetilde{\mathbf{A}}$ gerechnet wird. Da $\mathbf{A}$ wegen $\mathrm{Det}(\mathbf{A}) = -10^{-30}$ schon fast singulär ist, enthält $\widetilde{\mathbf{A}}$ wegen der Koeffizienten**intervalle** eine Punktmatrix, die tatsächlich singulär ist, so daß eine Lösung des Systems zwangsläufig unmöglich wird!

5. Beispiel

Mit Hilfe des langen Akkumulators und geeigneten Operatoren kann mit dem PASCAL–XSC–System ein Polynom **rundungsfehlerfrei** ausgewertet werden, wenn man **vorher** Argument und Polynomkoeffizienten in das jeweilige interne Zahlensystem des Rechners umgewandelt hat. Während die BCD–Version daher wegen fehlender Konversionsfehler **exakte** Polynomwerte liefern muß, sind Abweichungen der Ergebnisse bei der Binär–Version alleine auf die beschriebenen Konversionsfehler bei Argument und Polynomkoeffizienten zurückzuführen!

Mit dem ausmultiplizierten Polynom $\mathrm{P}(x) = \frac{1}{100} \cdot (3x - 1)^{13}$ und dem Argument $x_1 = 0.3333333333333333$ erhält man die folgenden Ergebnisse:

BCD–Version:	$\mathrm{P}(x_1) = -1.000000000000000 \cdot 10^{-210}$	**exakt!**
Binär–Version:	$\widetilde{\mathrm{P}}(\widetilde{x_1}) = -1.002225955526144 \cdot 10^{-15}$	

Beachten Sie bitte, daß in der Binärversion die Abweichung von **195 Größenordnungen** nur durch Konversionsfehler begründet ist, da die Polynomauswertung mit den binären Daten im langen Akkumulator rundungsfehlerfrei erfolgt! Weitere Informationen findet man dazu auf Seite 43.

Die obigen Beispiele zeigen recht eindrucksvoll den möglichen negativen Einfluß von Konversionsfehlern schon bei sehr einfachen numerischen Problemstellungen. Es soll an dieser Stelle aber ausdrücklich darauf hingewiesen werden, daß dieser Einfluß keine spezielle Eigenschaft etwa der Binärversion von PASCAL–XSC ist, denn man erhält z.B. mit TURBO–PASCAL und dem vergleichbaren Datentyp double für die Beispiele 1 bis 3 fast die gleichen Punktergebnisse wie mit dem PASCAL–XSC–System.

TURBO–PASCAL-Ergebnisse mit dem Datentyp double:

1. Beispiel: $\widetilde{a} \ominus \widetilde{b} = 1.776\ldots \cdot 10^{-15}$;

2. Beispiel: $\widetilde{\sin}(\widetilde{x_0}) = +3.6754\ldots \cdot 10^{-16}$;

3. Beispiel: $\widetilde{a} \odot \widetilde{b} \ominus \widetilde{c} \odot \widetilde{d} = +2.7408\ldots \cdot 10^{-16}$;

4. Beispiel: $\widetilde{x_1} = -4.5043\ldots \cdot 10^{15}$, $\widetilde{x_2} = +4.5043\ldots \cdot 10^{15}$;

5. Beispiel: $\widetilde{\mathrm{P}}(\widetilde{x_1}) = -9.59645977591760 \cdot 10^{-17}$;

Die TURBO–PASCAL–Lösungen der Beispiele 2 bis 5 sind im Vergleich zu den exakten Lösungen reine Phantasieergebnisse! Wertet man in Beispiel 4 die Ausdrücke für x_1, x_2 zusätzlich mit dem genaueren Datentyp extended aus, so scheitert die TURBO–PASCAL–Berechnung völlig, da die Koeffizientendeterminante im Nenner an Stelle von -10^{-30} mit 0 ausgewertet wird! Im Beispiel 5 kann man mit TURBO–PASCAL natürlich kein sinnvolles Ergebnis erwarten, da neben den Konversionsfehlern noch zusätzliche Auslöschungseffekte beim Horner-Schema zu berücksichtigen sind, da ein # Konzept in dieser Sprachumgebung nicht zur Verfügung steht!

Vergleicht man die PASCAL–XSC–Ergebnisse mit den entsprechenden Ergebnissen von TURBO–PASCAL bzgl. der Beispiele 1 bis 5, so erkennt man noch zusätzlich den großen Vorteil des PASCAL–XSC–Systems:

> Grobe Ergebniseinschließungen oder ein Programmabbruch mit Fehlermeldung des PASCAL–XSC–Systems weisen den Anwender darauf hin, daß der Algorithmus optimiert werden sollte, während die nur punktförmigen TURBO–PASCAL–Ergebnisse **keine Aussage** über die Zuverlässigkeit des Ergebnisses zulassen!

Zur **Vermeidung** der alleine durch Konversionsfehler bedingten Überschätzungen von Ergebnisintervallen gibt es grundsätzlich drei Möglichkeiten:

1. Man rechnet weiterhin mit der Binär–Version von PASCAL–XSC, jedoch mit Hilfe der Module mp_ari, mpi_ari in erhöhter Genauigkeit, wodurch die Laufzeiten jedoch drastisch anwachsen.
2. Man benutzt ein Algebra–System wie z.B. MATHEMATICA oder MAPLE. Die Nachteile sind hier jedoch die im Vergleich zu PASCAL–XSC schwerfälligen Programmiertechniken und vor allem die zu langen Laufzeiten.
3. Alle internen Rechnungen werden mit einer PASCAL–XSC–BCD–Version durchgeführt (BCD = Binary Coded Decimal), wodurch alle Datenkon-

versionen entfallen, vorausgesetzt, daß die Eingabedaten $a, b, \ldots$ nicht mehr Dezimalstellen enthalten, als das benutzte BCD–System zur Verfügung stellt.

Im Vergleich zu einer entsprechenden Binärversion ist ein BCD–System zwar merklich langsamer; wenn jedoch zur Vermeidung der beschriebenen Konversionsfehler mit den Modulen mp_ari bzw. mpi_ari in erhöhter Genauigkeit gerechnet wird oder wenn man ein Algebra–System benutzt, so hat man im Vergleich zum BCD–System doch merklich längere Laufzeiten und verliert mit einem Algebra–System zusätzlich die Mathematik bezogene einfache Programmierbarkeit eines PASCAL–XSC–Systems.

Zur Vermeidung der beschriebenen Konversionsfehler ist die BCD–Version von PASCAL–XSC daher ein geeignetes Hilfsmittel. Für die Beispiele 1,3,4 und 5 erhält man mit ihr die jeweils angegebenen **exakten Ergebnisse**, und im 2. Beispiel findet man für $\sin(x_0)$ jetzt die **optimale Einschließung:**

$$\sin(x_0) \in [\mathbf{-2.8461206985016149134}8\,;\ \mathbf{-2.8461206985016149134}7] \cdot 10^{-16}$$

Die BCD–Version von PASCAL–XSC bietet für mathematische Anwendungen folgende Vorteile:

1. Verfügbarkeit des Systems für Personal Computer, Workstations, Großrechner und Supercomputer mittels einer Implementierung in C.

2. Vermeidung von Konversionsfehlern, wodurch viele mathematische Problemstellungen erst zufriedenstellend lösbar werden!

3. Mit ergebnisverifizierenden Algorithmen werden gesicherte, mathematische Aussagen möglich, z.B. über die Existenz und Eindeutigkeit von Lösungen linearer oder nichtlinearer Gleichungen oder Gleichungssysteme.

4. Algorithmen der Binär–Version lassen sich ohne größeren Aufwand auf die BCD–Version übertragen.

5. Das Modul- und Operator-Konzept zusammen mit den implementierten Überladungen von Wertzuweisungen, Operatoren, Prozeduren und Funktionen ermöglichen die Realisierung gut lesbarer PASCAL–XSC–Programme.

6. Mit dem zusätzlichen Datentyp Longreal lassen sich alle Rechnungen im Bedarfsfall auch in erhöhter Genauigkeit durchführen.

7. Für die Datentypen real, Longreal, interval, Linterval, complex,Lcomplex, cinterval,Lcinterval steht eine Vielzahl von hochgenauen Standardfunktionen mit garantierten Fehlerschranken zur Verfügung.

8. Spezielle Funktionen der mathematischen Physik werden zusätzlich in hochgenauer Darstellung durch Hilfsmodule bereitgestellt.

9. Zur Realisierung hochgenauer Standard- oder spezieller Funktionen steht ein doppeltlanges real-Format mit $2 \cdot 13 = 26$ Mantissenstellen und ein nur durch die integer-Zahlen begrenzter Exponentenbereich zur Verfügung, zusammen

mit allen notwendigen Grundoperationen der Punkt- und Intervallrechnung für reelle und komplexe Zahlen. Zusätzlich existiert in diesem Zahlenformat eine Vielzahl von Standardfunktionen für reelle und komplexe Punkt- und Intervallrechnungen mit garantierten Fehlerschranken. Der Anwender kann Module mit Funktionen seiner eigenen Wahl erstellen und so das System beliebig erweitern.

10. Mit Hilfe der Toolbox–Bände I,II steht eine Vielzahl von Problemlösungen zur Verfügung, und an Hand der vielen Beispiele dieses Handbuches sollte der mit den Methoden des verifizierenden Rechnens vertraute Anwender in der Lage sein, Lösungsverfahren für eigene Aufgabenstellungen zu entwickeln.

Kapitel 3

BCD–Zahlenformate

3.1 Der Datentyp REAL

Gegeben sei die Dezimalzahl $x = m \cdot 10^{ex}$, dann gilt im Standard–Datenformat **REAL** für die normalisierten Zahlen:

$$m = 0 \qquad \text{oder} \qquad 0.1 \leq |m| \leq 0.9999999999999\,, \qquad \text{13 BCD–Ziffern}\,,$$

wobei für den Zehnerexponenten ex folgender Bereich zur Verfügung steht:

$$-254 \leq ex \leq +256\,; \qquad ex \in \{0, \pm 1, \pm 2, \dots\ \}$$

Bezeichnet man mit **NZ** das oben beschriebene normalisierte Zahlensystem, so gilt für alle positiven $x \in \mathbf{NZ}$ die Beziehung:

$$1.000000000000 \cdot 10^{-255} \leq x \leq 9.999999999999 \cdot 10^{+255}\,, \qquad \text{d.h.:}$$

MIN_REAL:= $1.000000000000 \cdot 10^{-255}$ ist die kleinste positive Zahl in **NZ**
MAXREAL:= $9.999999999999 \cdot 10^{+255}$ ist die größte (positive) Zahl in **NZ**

Bedeutet für $a, b \in \mathbf{NZ}$ z.B. $a \oplus b$ das Maschinenergebnis der Addition, so gilt i.a. $a \oplus b \neq a + b$, d.h. die Grundoperationen $+, -, *, /$ sind in **NZ nicht** abgeschlossen, so daß z.B. $(a + b) \notin \mathbf{NZ}$ zur nächsten Rasterzahl zu runden ist. Für die dabei auftretenden relativen Fehler ε_G gelten die folgenden Aussagen:

- Unter der Voraussetzung

  $$\text{MIN\_REAL} \leq |a \circ b| \leq \text{MAXREAL}, \qquad \text{mit:} \quad \circ \in \{+, -, *, /\}$$

  gilt z.B. für den relativen Fehler $\varepsilon_G := [a \oplus b - (a+b)]/(a+b)$ der Addition:

  $$|\varepsilon_G| \leq 0.5 \cdot 10^{-12} = \varepsilon(G): \quad \text{Fehlerschranke der Grundoperationen.}$$

- Unter der Voraussetzung $a, b \in \mathbf{NZ}$ und $a \circ b = 0$ ist auch stets das Maschinenergebnis gleich Null. Das exakte Ergebnis stimmt also mit dem Maschinenergebnis überein, wobei aber der relative Fehler ε_G jetzt nicht definiert ist!

- Unter der Voraussetzung $|a \circ b| > \text{MAXREAL}$, mit: $\circ \in \{+, -, *, /\}$ erfolgt eine Overflow–Meldung mit Programmabbruch.

Für den Datentyp **REAL** ist noch besonders zu beachten, daß es neben den normalisierten Zahlen auch noch die **denormalisierten** Zahlen gibt, die betragsmäßig kleiner als $\text{MIN\_REAL} = 10^{-255}$ sind. Dazu betrachten wir die folgenden Beispiele:

$$\begin{array}{rcl}
1\text{E}-255 \oslash 3\,\text{E}+000 & = & 3.333333333330\,\text{E}-256 \notin \mathbf{NZ} \\
1\text{E}-255 \oslash 3\,\text{E}+001 & = & 3.333333333300\,\text{E}-257 \notin \mathbf{NZ} \\
1\text{E}-255 \oslash 3\,\text{E}+002 & = & 3.333333333000\,\text{E}-258 \notin \mathbf{NZ} \\
 & \vdots & \\
1\text{E}-255 \oslash 3\,\text{E}+011 & = & 3.000000000000\,\text{E}-267 \notin \mathbf{NZ} \\
1\text{E}-255 \oslash 3\,\text{E}+012 & = & 0.000000000000\,\text{E}+000 \in \mathbf{NZ}
\end{array}$$

Man erkennt, daß mit kleiner werdendem Ergebnisexponenten die hinteren Mantissenstellen statt mit der Ziffer 3 nur noch mit Nullen besetzt werden, wodurch der relative Fehler ε_G jeweils um den Faktor 10 **anwächst!** Liegen die Ergebnisse x einer Operation daher im denormalisierten Bereich:

$$\text{MINREAL} = 10^{-267} \leq |x| < 10^{-255} = \text{MIN\_REAL}\,, \tag{3.1}$$

wobei die Besetzung der Mantisse mit von Null verschiedenen Ziffern nach obigen Beispielen entsprechend eingeschränkt ist, so kann für diese Grundoperation die im **NZ**–System geltende relative Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-12}$ nicht mehr garantiert werden!

> Hat man also für einen Algorithmus eine Fehlerabschätzung auf der Basis der Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-12}$ für die Grundoperationen vorgenommen, so dürfen keine Zwischen- oder Endergebnisse im denormalisierten Bereich (3.1) liegen, es sei denn, daß man sicher ist, daß die Fehlerschranken der Operationen mit Zahlen aus diesem kritischen Bereich nicht größer als $\varepsilon(G)$ sind!

Dazu einige Beispiele:

- $1 \oplus 1\,\text{E}-267 = 1$, d.h. der relative Fehler ist kleiner als $\varepsilon(G) = 0.5 \cdot 10^{-12}$.

Es ist zu beachten, daß im denormalisierten Bereich (3.1) und damit für alle Zahlen vom Typ REAL die Fehlerschranke für **Addition und Subtraktion** durch $\varepsilon(G) = 0.5 \cdot 10^{-12}$ gegeben ist!

Im denormalisierten Bereich haben alle Rasterzahlen im Gegensatz zu den normalisierten Zahlen den konstanten Abstand 10^{-267}, dessen Wert die kleinste positive Rasterzahl darstellt.

- $1\mathrm{E}{-}255 \oslash 3\mathrm{E}{+}011 = 3.000000000000\mathrm{E}{-}267$, d.h. der relative Fehler 0.1 ist jetzt größer als $\varepsilon(G)$.

- $1.11\mathrm{E}{-}255 \odot 3\mathrm{E}{-}010 = 3.330000000000\mathrm{E}{-}265$, d.h. der relative Fehler ist 0 und damit kleiner als $\varepsilon(G)$.

Bei **Multiplikation** und **Division** kann der Betrag des zugehörigen relativen Fehlers die Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-12}$ überschreiten, wenn das Ergebnis der Operation in den denormalisierten Bereich (3.1) fällt!

Es ist daher sinnvoll, für Multiplikation und Division Prozeduren und Funktionen bereitzustellen, die überprüfen, ob bei diesen Operationen die Fehlerschranken $\varepsilon(G) = 0.5 \cdot 10^{-12}$ bzw. $\varepsilon(G) = 1.0 \cdot 10^{-12}$ für die Rundung zur nächsten Gleitkommazahl bzw. für gerichtete Rundungen überschritten werden:

1. **function** MULNEXT_TEST(x,y: real; var error: boolean): real;

   Rundet das exakte Produkt $x \cdot y$ zur nächsten real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-12}$;

2. **function** MULUP_TEST(x,y: real; var error: boolean): real;

   Rundet das exakte Produkt $x \cdot y$ zur nächstgrößeren real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

3. **function** MULDOWN_TEST(x,y: real; var error: boolean): real;

   Rundet das exakte Produkt $x \cdot y$ zur nächstkleineren real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

4. **function** PRODUCT(m1,m2: real; ex1,ex2: integer;
   var error: boolean): real;

   Zu gegebenen real–Mantissen m1,m2 und den zugehörigen Zehnerexponenten ex1,ex2 berechnet die Funktion PRODUCT das zur nächsten real–Zahl gerundete Produkt $(m1 \cdot m2)^{ex1+ex2}$;
   Es muß gelten: $0.1 \leq |m1|, |m2| < 1$ oder $|m1|, |m2| = 0$.
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-12}$;

5. **function** ABC(a,b,c: real; var error: boolean): real;
   Zu gegebenen real–Zahlen a,b,c berechnet die Funktion ABC den Wert $(a \odot b) \odot c$ ohne vorzeitigen Overflow oder Underflow.
   error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

6. **function** ABCD(a,b,c,d: real; var error: boolean): real;
   Zu gegebenen real–Zahlen a,b,c,d berechnet die Funktion ABCD den Wert $(a \odot b) \odot (c \odot d)$ ohne vorzeitigen Overflow oder Underflow.
   error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

7. **function** DIVNEXT_TEST(x,y: real; var error: boolean): real;
   Rundet den exakten Quotienten x/y zur nächsten real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-12}$;

8. **function** DIVUP_TEST(x,y: real; var error: boolean): real;
   Rundet den exakten Quotienten x/y zur nächstgrößeren real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

9. **function** DIVDOWN_TEST(x,y: real; var error: boolean): real;
   Rundet den exakten Quotienten x/y zur nächstkleineren real–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

10. **function** QUOTIENT(m1,m2: real; ex1,ex2: integer;
    var error: boolean): real;
    Zu gegebenen real–Mantissen m1,m2 und den zugehörigen Zehnerexponenten ex1,ex2 berechnet die Funktion QUOTIENT den zur nächsten real–Zahl gerundeten Quotienten $(m1/m2)^{ex1-ex2}$;
    Es muß gelten: $0.1 \leq |m1|, |m2| < 1$ oder $|m1| = 0$.
    error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-12}$;

11. **function** ab_DIV_c(a,b,c: real; var error: boolean): real;
    Zu gegebenen real–Zahlen a,b,c berechnet die Funktion ab_DIV_c den Wert $(a \odot b) \oslash c$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

12. **function** a_DIV_bc(a,b,c: real; var error: boolean): real;
    Zu gegebenen real–Zahlen a,b,c berechnet die Funktion a_DIV_bc den Wert $a \oslash (b \odot c)$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

13. **function** ab_DIV_cd(a,b,c,d: real; var error: boolean): real;
    Zu den vorgegebenen real–Zahlen a,b,c,d berechnet die Funktion ab_DIV_cd den Wert $(a \odot b) \oslash (c \odot d)$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

3.1.1 Mathematische Konstanten

Dem Anwender stehen mehrere mathematische real–Konstanten zur Verfügung, die in Tabelle 3.1 zusammengestellt sind. Zusätzlich existiert im Longreal- und rrG-Format eine Vielzahl weiterer mathematischer Konstanten, die durch Aufruf entsprechender Funktionen benutzt werden können; vergl. dazu die Tabellen auf den Seiten 33,64;

Real – Konstanten			
Name	**Wert**	**Name**	**Wert**
MIN_EXP	-255	MIN_EXP_	-267
MIN_EXPL	-511	MIN_EXPL_	-531
MAX_EXP	$+255$	MAX_EXPL	$+511$
MANT_W	$+13$	MANT_WL	$+21$
TEST_26	-240	TEST_34	-232
PI_1	$3.141592653589 \cdot 10^{+00}$	PI_2	$7.932384626433 \cdot 10^{-13}$
PI_3	$8.327950288419 \cdot 10^{-26}$	PI_4	$7.169399375105 \cdot 10^{-39}$
PI_5	$8.209749445923 \cdot 10^{-52}$	PI_REST	$2.643383279503 \cdot 10^{-21}$
R_PI_R	$3.183098861838 \cdot 10^{-01}$	MIN_REAL	$1 \cdot 10^{-255}$

Tabelle 3.1: real–Konstanten.

Anmerkungen:

- Addiert man die Konstanten PI_1 bis PI_5 in den Akkumulator, so addiert man zu seinem Wert die $5 \cdot 13 = 65$–stellige, abgerundete Näherung für π mit der relativen Fehlerschranke $2.4881 \cdot 10^{-66}$.
- R_PI_R ist die zur nächsten real–Zahl aufgerundete Näherung für $1/\pi$.

3.1.2 Mathematische Standardfunktionen

Für das Argument x vom Typ **REAL** stehen mathematische Standardfunktionen zur Verfügung, die in Tabelle 3.2 zusammengestellt sind.

Alle mathematischen Funktionen aus Tabelle 3.2 sind mit Ausnahme von power(x,n,rndmode) **hochgenau**, d.h. das Maschinenergebnis unterscheidet sich vom optimal gerundeten Funktionsergebnis um maximal eine Einheit in der letzten Mantissenstelle. Diese Abweichung von 1 ulp tritt allerdings nur in **sehr seltenen Ausnahmefällen** auf!
Fallen Funktionswerte $y := f(x)$ in den Bereich (3.1), d.h. gilt:

$$0 \leq |y| < 10^{-255} = \text{MIN\_REAL},$$

Funktion	Aufruf	Funktion	Aufruf
$\|x\|$	abs(x)	x^y	power(x,y)
x^2	sqr(x)	$\sin(x)$	sin(x)
x^2-y^2	x2_y2(x,y)	$\cos(x)$	cos(x)
$\sqrt{x}$	sqrt(x), $x \geq 0$	$\tan(x)$	tan(x)
$\sqrt{1+x}-1$	sqrt1pm1(x); $x \geq -1$	$\cot(x)$	cot(x)
$\sqrt{1-x^2}$	sqrt1mx2(x); $\|x\| \leq 1$	$\sin(\pi x)$	sinpi(x)
$\sqrt{1+x^2}$	sqrt1px2(x)	$\cos(\pi x)$	cospi(x)
$\sqrt{x^2-1}$	sqrtx2m1(x); $\|x\| \geq 1$	$\tan(\pi x)$	tanpi(x)
$\sqrt{x^2+y^2}$	sqrtx2y2(x,y)	$\cot(\pi x)$	cotpi(x)
e^x	exp(x)	$\arcsin(x)$	arcsin(x)
e^x-1	expm1(x)	$\arccos(x)$	arccos(x)
e^{-x^2}	exp_x2(x)	$\arctan(x)$	arctan(x)
2^x	exp2(x)	$\arctan(x/y)$	arctan2(x,y)
10^x	exp10(x)	$\mathrm{arccot}(x)$	arccot(x)
$\ln(x)$	ln(x); $x>0$	$\sinh(x)$	sinh(x)
$\ln(e \cdot x)$	ln_ex(x)	$\cosh(x)$	cosh(x)
$\ln(1+x)$	ln1p(x); $x>-1$	$\tanh(x)$	tanh(x)
$0.5 \cdot \ln(x^2+y^2)$	ln_sqrtx2y2(x,y)	$\coth(x)$	coth(x)
$0.5 \cdot \ln[(1+x)^2+y^2]$	ln_sqrt1px2y2(x,y)	$\mathrm{arsinh}(x)$	arsinh(x)
$\log_a(x)$	loga(a,x)	$\mathrm{arcosh}(x)$	arcosh(x)
$\log_2(x)$	log2(x); $x>0$	$\mathrm{arcosh}(1+x)$	arcosh1p(x)
$\log_{10}(x)$	log10(x); $x>0$	$\mathrm{artanh}(x)$	artanh(x)
$\log_{10}(1+x)$	log1p(x); $x>-1$	$\mathrm{arcoth}(x)$	arcoth(x)
x^n	power$(x,n,rndmode)$		

Tabelle 3.2: Hochgenaue Funktionen vom Typ real; a,x,y: real–Ausdrücke;

und werden hintere, von Null verschiedene Ziffern der Ergebnismantisse gleich Null gesetzt oder entsteht Underflow, so daß die Funktionsergebnisse dadurch nicht mehr hochgenau sind, so erfolgt die Fehlermeldung:

rel. error exceeds error bound

mit Programmabbruch.

Anmerkungen zu den Standardfunktionen:

1. Mit Ausnahme von power(x,n,rndmode) zur Berechnung von Unter- bzw. Oberschranken (rndmode = -1 bzw.$+1$) für $x^n, n = 0, 1, 2 \ldots$ haben alle Funktionen aus Tabelle 3.2 die relative Fehlerschranke $5.0000001 \cdot 10^{-12}$, d.h. wenn etwa $\exp(x)$ das i.a. fehlerbehaftete Maschinenergebnis der Exponentialfunktion zum Argument x: real bedeutet, so gilt:
$$\exp(x) = e^x \cdot (1 + \varepsilon), \quad \text{mit} \quad |\varepsilon| < 5.0000001 \cdot 10^{-12}$$
2. Sei $x = 1.1111 \cdot 10^{-130}$; Während das Produkt $x \odot x$ mit einem relativen Fehler $> 0.5 \cdot 10^{-12}$ ohne Fehlermeldung in den Bereich (3.1) fällt, liefert $y := \mathrm{SQR}(x)$ eine Fehlermeldung, da die Quadratfunktion $\mathrm{SQR}(x)$ für obiges Argument x nicht mehr hochgenau ausgewertet werden kann.
3. Bei den trigonometrischen Funktionen $\sin(x), \cos(x), \tan(x), \cot(x)$ ist das Argument x eingeschränkt auf den Bereich (Polstellen ausgenommen):
$$|x| < 10^8\,;$$
4. Bei den Funktionen $\tan(\pi \cdot x)$ und $\cot(\pi \cdot x)$ ist das Argument x eingeschränkt auf den Bereich (Polstellen ausgenommen):
$$|x| \leq 1.0737418235 \cdot 10^9\,;$$
5. Bei den Funktionen $\sin(\pi \cdot x)$ und $\cos(\pi \cdot x)$ ist das Argument x eingeschränkt auf den Bereich:
$$|x| \leq 2.147483647 \cdot 10^9\,;$$
6. Argumente außerhalb der drei obigen Bereiche erzeugen bei den entsprechenden Funktionen eine Fehlermeldung. Verletzungen des Definitionsbereiches, z.B. bei der $\sqrt{x}$–Funktion oder Argumente im Overflowbereich erzeugen ebenfalls Fehlermeldungen mit Programmabbruch.

3.1.3 Funktionen, Prozeduren und Operatoren

In der folgenden Auflistung werden alle diejenigen Funktionen, Prozeduren und Operatoren angegeben, die mit den Datentypen **integer,real** in Verbindung stehen.

Wie im Binär–System existieren auch in der PASCAL–XSC–BCD–Version die Vergleichsoperatoren $= \; <> \; < \; <= \; > \; >=$ für Operanden vom Typ real und integer.

Für die Grundoperationen $+ \; - \; * \; /$ mit Rundung der exakten Ergebnisse zur nächsten real-Zahl einschließlich gerichteter Rundungen stehen alle arithmetischen Operatoren für real- und integer-Operanden zur Verfügung und werden hier nicht weiter beschrieben.

1. **function** CEIL (b: real): real;
   y:=CEIL(x) liefert die kleinste ganze real–Zahl y, die größer oder gleich x ist: $x \leq y$;
2. **function** FLOOR (b: real): real;
   y:=FLOOR(x) liefert die größte ganze real–Zahl y, die kleiner oder gleich x ist: $y \leq x$;
3. **function** FRAC (b: real): real;
   y:=FRAC(x) liefert den Nachkommawert von x;
   y:=FRAC(-12.012) liefert: $y = -1.2\text{E}{-002}$;
   $0 \leq |y| \leq 0.9999999999999$;
4. **function** ROUND (b: real): integer;
   n:=ROUND(x) liefert die zu x nächstgelegene integer–Zahl n;
   n:=ROUND(3.5) liefert: $y = 4$
   integer–Bereich: $-2147483648 \leq n \leq +2147483647$;
5. **function** ROUND (b: real): real;
   y:=ROUND(x) liefert die zu x nächstgelegene, ganze real–Zahl y;
   y:=ROUND(3.5) liefert: $y = 4$
   y–Bereich: $-2147483648 \leq y \leq +2147483647$;
6. **function** TRUNC (b: real): integer;
   n:=TRUNC(x) liefert die integer–Zahl n, die durch Abschneiden der Nachkommastellen von x entsteht.
   integer–Bereich: $-2147483648 \leq n \leq +2147483647$;
7. **function** TRUNC (b: real): real;
   y:=TRUNC(x) liefert die ganze real–Zahl y, die durch Abschneiden der Nachkommastellen von x entsteht.
   y–Bereich: $-2147483648 \leq y \leq +2147483647$;
8. **function** SIGN (b: real): integer;
   y:=SIGN(x) liefert das Vorzeichen von x; $y \in \{-1, 0, +1\}$;
9. **function** MANT (b: real): real;
   m:=MANT(x) liefert die vorzeichenbehaftete Mantisse von x;
   es gilt: $m = 0$ oder $0.1 \leq |m| < 1$;
10. **function** EXPO (b: real): integer;
    ex:=EXPO(x) liefert den Zehnerexponenten ex bzgl. der Darstellung: $x = m \cdot 10^{ex}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: $m = 0$ oder $0.1 \leq |m| < 1$;
    Für $x \neq 0$ gilt: $-266 \leq ex \leq +256$;
    Im Fall $x = 0$ gilt: $ex = -2147483647$;

11. **function** EXPO_0 (b: real): integer;
    ex:=EXPO_0(x) liefert den Zehnerexponenten ex bzgl. der Darstellung: $x = m \cdot 10^{ex}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: m= 0 oder $0.1 \leq |m| < 1$;
    Für $x \neq 0$ gilt: $-266 \leq ex \leq +256$;
    Im Falle $x = 0$ gilt: $ex = 0$, im Gegensatz zur EXPO–Funktion!

12. **function** EXPO_ADD (a,b: integer): integer;
    s := EXPO_ADD(a,b) liefert die Summe $s := a + b$;
    integer–Unterlauf: $s := -2147483647$;
    integer–Überlauf: Fehlermeldung;

13. **procedure** INTEGER_ADD (a,b: integer; var s,k: integer);
    INTEGER_ADD(a,b,s,k) liefert die Summe $s := a + b$;
    integer–Unterlauf: $k := -1$;
    integer–Überlauf: $k := +1$;
    $s := a + b$ wird exakt berechnet: $k := 0$;

14. **function** COMP (m: real; ex: integer): real;
    y:=COMP(m,ex) liefert die real–Zahl y, zusammengesetzt aus der Mantisse m und dem Zehnerexponenten ex bzgl. der Darstellung: $x = m \cdot 10^{ex}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: m= 0 oder $0.1 \leq |m| < 1$;
    Es muß gelten: $-266 \leq ex \leq +256$, und im Bereich $-266 \leq ex \leq -255$ muß die Mantisse m, wie auf S. 12 beschrieben, zusätzlich eingeschränkt sein!
    Beispiele:
    y:=COMP$(0.1, -267) \longrightarrow$ y = 0; „Exponent too small“–Meldung; kein Programmabbruch!
    y:=COMP$(0.12\ldots23, -255) \longrightarrow$ y = $1.23\ldots\mathbf{20}\text{E} - 256$;
    „Mantissa bits lost...“–Meldung; kein Programmabbruch!
    y:=COMP(Long(0.1), +257) $\longrightarrow$ „Overflow–Meldung“ mit Programmabbruch.

15. **function** PRED (b: real): real;
    y:=PRED(x) liefert bzgl. x die nächstkleinere Maschinenzahl y.

16. **function** SUCC (b: real): real;
    y:=SUCC(x) liefert bzgl. x die nächstgrößere Maschinenzahl y.

17. **function** RVAL (s: string): real;
    y:=RVAL(s) wandelt den String s in eine real–Zahl y. Der String s muß eine Zeichenfolge enthalten, die eine real–Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird durch Runden zur nächsten real–Zahl berücksichtigt.

18. **function** RVAL (s: string; round: integer): real;
    y:=RVAL(s,round) wandelt den String s in eine real–Zahl y. Der String s muß eine Zeichenfolge enthalten, die eine real–Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird durch Runden gemäß round berücksichtigt:

$$\text{round} = \left\{ \begin{array}{rl} -1 & \text{nach unten gerundet} \\ 0 & \text{nächstliegend gerundet} \\ +1 & \text{nach oben gerundet} \end{array} \right\}$$

19. **function** IMAGE (s: real): string;
    s:=IMAGE(y) wandelt die real–Zahl y in einen String s entsprechend der Standardausgabe von real–Werten.
    Beispiel:
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-7)$ liefert: $\mathrm{s} = -2.4\mathrm{E}-007\,;$

20. **function** IMAGE (s: real; width: integer): string;
    s:=IMAGE(y,w) wandelt die real–Zahl y in einen String s aus mindestens w Zeichen entsprechend der Ausgabe von real–Werten, eventuell mit Auffüllen von Nullen am Mantissenende.
    Beispiele:
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-7,0) \longrightarrow \mathrm{s} = -2.4\mathrm{E}-007\,;$
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-7,11) \longrightarrow \mathrm{s} = -2.350\mathrm{E}-007\,;$

21. **function** IMAGE (s: real; width, fracs: integer): string;
    s:=IMAGE(y,w,f) wandelt die real–Zahl y in einen String s aus wenigstens w Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) entsprechend der Ausgabe von real–Werten in Festkommadarstellung mit f Nachkommastellen, wobei Nullen angehängt werden, wenn f unnötig groß ist.
    Beispiele:
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,12,7) \longrightarrow \mathrm{s} = \sqcup\sqcup-0.0023500\,;$
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,7,4) \longrightarrow \mathrm{s} = -0.0024\,;$
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,3,4) \longrightarrow \mathrm{s} = -0.0024\,;$

22. **function** IMAGE (s: real; width, fracs, round: integer): string;
    s:=IMAGE(y,w,f,round) wandelt die real–Zahl y in einen String s aus wenigstens w Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) entsprechend der Ausgabe von real–Werten in Festkommadarstellung mit f Nachkommastellen, wobei Nullen angehängt werden, wenn f unnötig groß ist. Bei zu kleinem f wird entsprechend round gerundet, vergl. die entsprechende Funktion LVAL.
    Beispiele:
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,7,4,+0) \longrightarrow \mathrm{s} = -0.0024\,;$
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,3,4,+1) \longrightarrow \mathrm{s} = -0.0023\,;$
    $\mathrm{s} := \mathrm{IMAGE}(-2.35\,,\mathrm{E}-3,7,4,-1) \longrightarrow \mathrm{s} = -0.0024\,;$

23. **function** ENTIRE (x: real; var gerade: boolean): boolean;
    bl:=ENTIRE(x, gerade) liefert an bl den Wert true, wenn die Real–Zahl x eine **ganze** Zahl ist, sonst erhält bl den Wert false.
    Falls x ganzzahlig ist, erhält gerade den Wert true (false), wenn x zusätzlich gerade (ungerade) ist.
    Ist x nicht ganzzahlig, so bleibt die Variable gerade bedeutungslos.

24. **procedure** MANT_EXPO_PRODUCT(F1,F2: real; var MP: real;
    var exp: integer);
    MANT_EXPO_PRODUCT liefert zu den real–Zahlen F1,F2 die zur nächsten real–Zahl gerundete Mantisse MP des exakten Produkts $\mathrm{F1} \cdot \mathrm{F2}$ und den zugehörigen Zehnerexponenten exp.
    Im Fall $\mathrm{F1} \cdot \mathrm{F2} = 0$ gilt $\mathrm{MP} = 0$ und $\mathrm{exp} = -\mathrm{maxint} = -2147483647$, und im Fall $\mathrm{F1} \cdot \mathrm{F2} \neq 0$ gilt: $0.1 \leq |\mathrm{MP}| < 1$.

25. **priority M_10HOCH** $= *$;
    operator M_10HOCH (x: real; k: integer) rint: real;
    y := x M_10HOCH k berechnet zur real–Zahl x das Produkt $\mathrm{y} = \mathrm{x} \cdot 10^k$, $k \in \{0, \pm 1, \pm 2, \dots\}$;
    Im Fall $10^{-267} \leq |\mathrm{x} \cdot 10^k| < 10^{-255}$ wird ohne Fehlermeldung zur nächsten Rasterzahl gerundet, wenn das Ergebnis wegen einer zu langen Mantisse von x nicht mehr exakt dargestellt werden kann.
    Im Fall $0 < |\mathrm{x} \cdot 10^k| < 10^{-267}$ wird das Ergebnis auf Null gesetzt.
    Der Operand x darf auch vom Typ integer sein!

3.2 Der Datentyp LONGREAL

Durch das Modul STDMOD wird der Datentyp **LONGREAL** zur Verfügung gestellt. Gegeben sei die Dezimalzahl $x = m \cdot 10^{ex}$, dann gilt im **LONGREAL**–Datenformat für die normalisierten Zahlen:

$$m = 0 \qquad \text{oder} \qquad 0.1 \leq |m| \leq 0.999999999999999999999\,, \qquad \text{21 BCD–Ziffern}\,;$$

wobei für den Zehnerexponenten ex folgender Bereich zur Verfügung steht:

$$-510 \leq ex \leq +512\,; \qquad ex \in \{0, \pm 1, \pm 2, \ldots\ \}$$

Bezeichnet man mit **NZ** das oben beschriebene normalisierte Zahlensystem, so gilt für alle positiven $x \in \mathbf{NZ}$ die Beziehung:

$$1.00000000000000000000 \cdot 10^{-511} \leq x \leq 9.99999999999999999999 \cdot 10^{+511}\,, \qquad \text{d.h.:}$$

MINLONGREAL:= $1.0\ldots0 \cdot 10^{-511}$ ist die kleinste positive Zahl in **NZ**
MAXLONGREAL:= $9.9\ldots9 \cdot 10^{+511}$ ist die größte (positive) Zahl in **NZ**

idxminLongreal (Longreal–Konstante) idxmaxLongreal (Longreal–Konstante) Bedeutet für $a, b \in \mathbf{NZ}$ z.B. $a \oplus b$ das Maschinenergebnis der Addition, so gilt i.a. $a \oplus b \neq a + b$, d.h. die Grundoperationen $+, -, *, /$ sind in **NZ nicht** abgeschlossen, so daß z.B. $(a + b) \notin \mathbf{NZ}$ zur nächsten Rasterzahl zu runden ist. Für die dabei auftretenden relativen Fehler ε_G gelten die folgenden Aussagen:

- Unter der Voraussetzung

  $$\text{MINLONGREAL} \leq |a \circ b| \leq \text{MAXLONGREL}, \qquad \text{mit:} \quad \circ \in \{+, -, *, /\}$$

  gilt z.B. für den relativen Fehler $\varepsilon_G := [a \oplus b - (a+b)]/(a+b)$ der Addition:

  $$|\varepsilon_G| \leq 0.5 \cdot 10^{-20} = \varepsilon(G)\,: \quad \text{Fehlerschranke der Grundoperationen.}$$

- Unter der Voraussetzung $a, b \in \mathbf{NZ}$ und $a \circ b = 0$ ist auch stets das Maschinenergebnis gleich Null. Das exakte Ergebnis stimmt also mit dem Maschinenergebnis überein, wobei aber der relative Fehler ε_G jetzt nicht definiert ist!

- Unter der Voraussetzung

  $$|a \circ b| > \text{MAXLONGREAL}, \qquad \text{mit:} \quad \circ \in \{+, -, *, /\}$$

  erfolgt eine Overflow–Meldung mit Programmabbruch.

Für den Datentyp **LONGREAL** ist noch besonders zu beachten, daß es neben den normalisierten Zahlen auch noch die **denormalisierten** Zahlen gibt, die betragsmäßig kleiner als MINLONGREAL $= 10^{-511}$ sind. Dazu betrachten wird die folgenden Beispiele:

$$\begin{array}{lcl}
1\mathrm{E}-511 \oslash 3\,\mathrm{E}+000 & = & 3.33333333333333333330\,\mathrm{E}-512 \notin \mathbf{NZ} \\
1\mathrm{E}-511 \oslash 3\,\mathrm{E}+001 & = & 3.33333333333333333300\,\mathrm{E}-513 \notin \mathbf{NZ} \\
1\mathrm{E}-511 \oslash 3\,\mathrm{E}+002 & = & 3.33333333333333333000\,\mathrm{E}-514 \notin \mathbf{NZ} \\
 & \vdots & \\
1\mathrm{E}-511 \oslash 3\,\mathrm{E}+019 & = & 3.00000000000000000000\,\mathrm{E}-531 \notin \mathbf{NZ} \\
1\mathrm{E}-511 \oslash 3\,\mathrm{E}+020 & = & 0.00000000000000000000\,\mathrm{E}+000 \in \mathbf{NZ}
\end{array}$$

Man erkennt, daß mit kleiner werdendem Ergebnisexponenten die hinteren Mantissenstellen statt mit der Ziffer 3 nur noch mit Nullen besetzt werden, wodurch der relative Fehler ε_G jeweils um den Faktor 10 **anwächst!** Liegen die Ergebnisse x einer Operation daher im denormalisierten Bereich

$$\text{(3.2)} \qquad \text{MIN\_LONGREAL} = 10^{-531} \leq |x| < 10^{-511} = \text{MINLONGREAL}\,,$$

wobei die Besetzung der Mantisse mit von Null verschiedenen Ziffern nach obigen Beispielen entsprechend eingeschränkt ist, so kann für alle Grundoperationen die im **NZ**–System geltende Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-20}$ nicht mehr garantiert werden!

Hat man also für einen Algorithmus eine Fehlerabschätzung auf der Basis der Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-20}$ für die Grundoperationen vorgenommen, so dürfen keine Zwischen- oder Endergebnisse im denormalisierten Bereich (3.2) liegen, es sei denn, daß man sicher ist, daß die Fehlerschranken der Operationen mit Zahlen aus diesem kritischen Bereich nicht größer als $\varepsilon(G)$ sind!

Dazu einige Beispiele:

- $1 \oplus 1\,\mathrm{E}-531 = 1$, d.h. der relative Fehler ist kleiner als $\varepsilon(G) = 0.5 \cdot 10^{-20}$.

  Es ist zu beachten, daß im denormalisierten Bereich (3.2) und damit für alle Zahlen vom Typ LONGREAL die Fehlerschranke für **Addition und Subtraktion** durch $\varepsilon(G) = 0.5 \cdot 10^{-20}$ gegeben ist!

  Im denormalisierten Bereich haben alle Rasterzahlen im Gegensatz zu den normalisierten Zahlen den konstanten Abstand 10^{-531}, dessen Wert die kleinste positive Rasterzahl darstellt.

- $1\mathrm{E}-511 \oslash 3\mathrm{E}+019 = 3.000\ldots000\mathrm{E}-531$, d.h. der relative Fehler 0.1 ist jetzt größer als $\varepsilon(G)$.

- $1.11\mathrm{E}-511 \odot 3\mathrm{E}-010 = 3.330000000000\mathrm{E}-521$, d.h. der relative Fehler ist 0 und damit kleiner als $\varepsilon(G)$.

> Bei **Multiplikation** und **Division** kann der Betrag des zugehörigen relativen Fehlers die Fehlerschranke $\varepsilon(G) = 0.5 \cdot 10^{-20}$ überschreiten, wenn das Ergebnis der Operation in den denormalisierten Bereich (3.2) fällt!

Es ist daher sinnvoll, für Multiplikation und Division Prozeduren und Funktionen bereitzustellen, die überprüfen, ob bei diesen Operationen die Fehlerschranken $\varepsilon(G) = 0.5 \cdot 10^{-20}$ bzw. $\varepsilon(G) = 1.0 \cdot 10^{-20}$ für die Rundung zur nächsten Gleitkommazahl bzw. für gerichtete Rundungen überschritten werden:

1. **function** MULNEXT_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet das exakte Produkt $x \cdot y$ zur nächsten Longreal–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-20}$;

2. **function** MULUP_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet das exakte Produkt $x \cdot y$ zur nächstgrößeren Longreal–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-20}$;

3. **function** MULDOWN_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet das exakte Produkt $x \cdot y$ zur nächstkleineren Longreal–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $\geq 1 \cdot 10^{-20}$;

4. **function** PRODUCT(m1,m2: Longreal; ex1,ex2: integer; var error: boolean): Longreal;
   Zu gegebenen Longreal–Mantissen m1,m2 und den zugehörigen Zehnerexponenten ex1,ex2 berechnet die Funktion PRODUCT das zur nächsten Longreal–Zahl gerundete Produkt $(m1 \cdot m2)^{ex1+ex2}$;
   Es muß gelten: $0.1 \leq |m1|, |m2| < 1$ oder $|m1|, |m2| = 0$.
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-20}$;

5. **function** ABC(a,b,c: Longreal; var error: boolean): Longreal;
   Zu gegebenen Longreal–Zahlen a,b,c berechnet die Funktion ABC den Wert $(a \odot b) \odot c$ ohne vorzeitigen Overflow oder Underflow.
   error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

6. **function** ABCD(a,b,c,d: Longreal; var error: boolean): Longreal;
   Zu gegebenen Longreal–Zahlen a,b,c,d berechnet die Funktion ABCD den Wert $(a \odot b) \odot (c \odot d)$ ohne vorzeitigen Overflow oder Underflow.
   error = TRUE $\leadsto$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

7. **function** DIVNEXT_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet den exakten Quotienten x/y zur nächsten Longreal–Zahl;
   error = TRUE $\leadsto$ relativer Fehler $> 0.5 \cdot 10^{-20}$;

8. **function** DIVUP_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet den exakten Quotienten x/y zur nächstgrößeren Longreal–Zahl;
   error = TRUE $\rightsquigarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$;

9. **function** DIVDOWN_TEST(x,y: Longreal; var error: boolean): Longreal;
   Rundet den exakten Quotienten x/y zur nächstkleineren Longreal–Zahl;
   error = TRUE $\rightsquigarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$;

10. **function** QUOTIENT(m1,m2: Longreal; ex1,ex2: integer;
    var error: boolean): Longreal;
    Zu gegebenen Longreal–Mantissen m1,m2 und den zugehörigen Zehnerexponenten ex1,ex2 berechnet die Funktion QUOTIENT den zur nächsten Longreal–Zahl gerundeten Quotienten $(m1/m2)^{ex1-ex2}$;
    Es muß gelten: $0.1 \leq |m1|, |m2| < 1$ oder $|m1| = 0$.
    error = TRUE $\rightsquigarrow$ relativer Fehler $> 0.5 \cdot 10^{-20}$;

11. **function** ab_DIV_c(a,b,c: Longreal; var error: boolean): Longreal;
    Zu den vorgegebenen Longreal–Zahlen a,b,c berechnet die Funktion ab_DIV_c den Wert $(a \odot b) \oslash c$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\rightsquigarrow$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

12. **function** a_DIV_bc(a,b,c: Longreal; var error: boolean): Longreal;
    Zu den vorgegebenen Longreal–Zahlen a,b,c berechnet die Funktion a_DIV_bc den Wert $a \oslash (b \odot c)$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\rightsquigarrow$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

13. **function** ab_DIV_cd(a,b,c,d: Longreal; var error: boolean): Longreal;
    Zu den vorgegebenen Longreal–Zahlen a,b,c,d berechnet die Funktion ab_DIV_cd den Wert $(a \odot b) \oslash (c \odot d)$ ohne vorzeitigen Overflow oder Underflow.
    error = TRUE $\rightsquigarrow$ Im denormalisierten Bereich ist eine eigene Fehlerabschätzung notwendig!

3.2.1 Standardoperatoren

Für integer–, real– und Longreal–Operanden stehen alle in **PASCAL-XSC** üblichen Vergleichsoperatoren zur Verfügung:

$$= \quad <> \quad > \quad >= \quad < \quad <=$$

Der rechte oder linke Vergleichsoperand kann dabei wahlweise ein integer–, real– oder Longrealausdruck sein, d.h. möglich ist z.B.:

$$3 + \sin(4.1) <= \mathrm{LONG}(-4.33) - 2.1113\mathrm{E}{-13}$$

Für alle Kombinationen aus integer–, real– und Longreal–Operanden stehen die arithmetischen Operatoren

$$+, -, *, / \quad \text{mit den gerichteten Rundungen} \;\; +>, +<, ->, -<, *>, *<, />, /<$$

zur Verfügung.

3.2.2 Mathematische Standardfunktionen

Für das Argument x vom Typ **LONGREAL** stehen hochgenaue mathematische Standardfunktionen zur Verfügung, die in nachfolgender Tabelle 3.3 zusammengestellt sind. Mit Ausnahme von power(x,n,rndmode) zur Berechnung von Unter- bzw. Oberschranken (rndmode = -1 bzw.+1) für $x^n, n = 0, 1, 2\ldots$ besitzen alle Funktionen die relative Fehlershranke $\varepsilon(f) = 5.0078 \cdot 10^{-21}$.

Definition des relativen Fehlers:
Bedeutet $\widetilde{f}(x)$ das fehlerbehaftete Maschinenergebnis des exakten Funktionswertes $f(x)$, so gilt für den relativen Fehler ε_f:

$$\varepsilon_f := \frac{\widetilde{f}(x) - f(x)}{f(x)}, \quad f(x) \neq 0\,; \qquad |\varepsilon_f| \leq \varepsilon(f) \;:\; \text{Rel. Fehlerschranke für } f(x)\,;$$

Im Fall $f(x) = 0$ gilt für alle Standardfunktionen stets $\widetilde{f}(x) = 0$, so daß obige Einschränkung $f(x) \neq 0$ keine praktische Bedeutung hat!

Fallen Funktionswerte $y := f(x)$ in den Bereich (3.2), d.h. gilt:

$$0 \leq |y| < 10^{-511} = \text{MINLONGREAL},$$

und werden hintere, von Null verschiedene Ziffern der Ergebnismantisse gleich Null gesetzt oder entsteht Underflow, so daß die angegebenen Fehlerschranken nicht mehr garantiert werden können, so erfolgt die Fehlermeldung:

rel. error exceeds error bound

mit Programmabbruch.

Funktion	Aufruf	Funktion	Aufruf
$\|x\|$	abs(x)	x^y	power(x,y)
x^2	sqr(x)	$\sin(x)$	sin(x)
$x^2 - y^2$	x2_y2(x,y)	$\cos(x)$	cos(x)
$\sqrt{x}$	sqrt(x), $x \geq 0$	$\tan(x)$	tan(x)
$\sqrt{1+x}-1$	sqrt1pm1(x); $x \geq -1$	$\cot(x)$	cot(x)
$\sqrt{1-x^2}$	sqrt1mx2(x); $\|x\| \leq 1$	$\sin(\pi x)$	sinpi(x)
$\sqrt{1+x^2}$	sqrt1px2(x)	$\cos(\pi x)$	cospi(x)
$\sqrt{x^2-1}$	sqrtx2m1(x); $\|x\| \geq 1$	$\tan(\pi x)$	tanpi(x)
$\sqrt{x^2+y^2}$	sqrtx2y2(x,y)	$\cot(\pi x)$	cotpi(x)
e^x	exp(x)	$\arcsin(x)$	arcsin(x)
$e^x - 1$	expm1(x)	$\arccos(x)$	arccos(x)
e^{-x^2}	exp_x2(x)	$\arctan(x)$	arctan(x)
2^x	exp2(x)	$\arctan(x/y)$	arctan2(x,y)
10^x	exp10(x)	$\text{arccot}(x)$	arccot(x)
$\ln(x)$	ln(x); $x > 0$	$\sinh(x)$	sinh(x)
$\ln(e \cdot x)$	ln_ex(x)	$\cosh(x)$	cosh(x)
$\ln(1+x)$	ln1p(x); $x > -1$	$\tanh(x)$	tanh(x)
$0.5 \cdot \ln(x^2+y^2)$	ln_sqrtx2y2(x,y)	$\coth(x)$	coth(x)
$0.5 \cdot \ln[(1+x)^2+y^2]$	ln_sqrt1px2y2(x,y)	$\text{arsinh}(x)$	arsinh(x)
$\log_a(x)$	loga(a,x)	$\text{arcosh}(x)$	arcosh(x)
$\log_2(x)$	log2(x); $x > 0$	$\text{arcosh}(1+x)$	arcosh1p(x)
$\log_{10}(x)$	log10(x); $x > 0$	$\text{artanh}(x)$	artanh(x)
$\log_{10}(1+x)$	log1p(x); $x > -1$	$\text{arcoth}(x)$	arcoth(x)
x^n	power($x, n, rndmode$)		

Tabelle 3.3: Hochgenaue Funktionen vom Typ Longreal; a,x,y: Longreal;

Anmerkungen zu den Standardfunktionen:

1. Bei den trigonometrischen Funktionen $\sin(x), \cos(x), \tan(x), \cot(x)$ ist das Argument x eingeschränkt auf den Bereich (Polstellen ausgenommen):

$$|x| < 10^{+8}$$

2. Bei den Funktionen $\tan(\pi \cdot x)$ und $\cot(\pi \cdot x)$ ist das Argument x eingeschränkt auf den Bereich (Polstellen ausgenommen):
$$|x| \leq 1.0737418235 \cdot 10^9 \,;$$
3. Bei den Funktionen $\sin(\pi \cdot x)$ und $\cos(\pi \cdot x)$ ist das Argument x eingeschränkt auf den Bereich:
$$|x| \leq 2.147483647 \cdot 10^9 \,;$$
4. Argumente außerhalb der drei obigen Bereiche erzeugen bei den entsprechenden Funktionen eine Fehlermeldung. Verletzungen des Definitionsbereiches, z.B. bei der $\sqrt{x}$–Funktion oder Argumente im Overflowbereich erzeugen ebenfalls Fehlermeldungen mit Programmabbruch.

3.2.3 Hilfsfunktionen und Operatoren

Die folgenden Funktionen stehen mit dem Datentyp **LONGREAL** in Verbindung:

1. **function** CEIL (b: Longreal): Longreal;
   y:=CEIL(x) liefert die kleinste ganze Longreal–Zahl y, die größer oder gleich x ist: x ≤ y;
2. **function** FLOOR (b: Longreal): Longreal;
   y:=FLOOR(x) liefert die größte ganze Longreal–Zahl y, die kleiner oder gleich x ist: y ≤ x;
3. **function** FRAC (b: Longreal): Longreal;
   y:=FRAC(x) liefert den Nachkommawert von x;
   y:=FRAC(−12.012) liefert: y = −1.2E−002 ;
   $0 \leq |y| \leq 0.99999999999999999999$;
4. **function** LONG (b: real): Longreal;
   y:=LONG(x) liefert die zu x wertgleiche Longreal–Zahl y;
5. **function** LONG (b: integer): Longreal;
   y:=LONG(n) liefert die zu n wertgleiche Longreal–Zahl y;
6. **function** ROUND (b: Longreal): integer;
   n:=ROUND(x) liefert die zu x nächstgelegene integer–Zahl n;
   n:=ROUND(LONG(3.5)) liefert: y = 4
   integer–Bereich: −2147483648 ≤ n ≤ +2147483647 ;
7. **function** ROUND (b: Longreal): Longreal;
   y:=ROUND(x) liefert die zu x nächstgelegene, ganze Longreal–Zahl y; y:=ROUND(LONG(3.5)) liefert: y = 4
   y–Bereich: −2147483648 ≤ y ≤ +2147483647 ;

8. **function** SHORT (b: Longreal): real;
   y:=SHORT(x) rundet x zur nächstgelegenen real–Zahl y;

9. **function** SHORTDOWN (b: Longreal): real;
   y:=SHORTDOWN(x) rundet x zur nächstkleineren real–Zahl y;

10. **function** SHORTUP (b: Longreal): real;
    y:=SHORTUP(x) rundet x zur nächstgrößeren real–Zahl y;

11. **function** SHORT_TEST (x: Longreal; var error: boolean): real;
    $y :=$ SHORT_TEST$(x, error)$ rundet x zur nächsten real–Zahl y;
    error = TRUE $\Longleftrightarrow$ relativer Fehler $> 0.5 \cdot 10^{-12}$;

12. **function** SHORTUP_TEST (x: Longreal; var error: boolean): real;
    $y :=$ SHORTUP_TEST$(x, error)$ rundet x zur nächstgrößeren real–Zahl y;
    error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

13. **function** SHORTDOWN_TEST (x: Longreal; var error: boolean): real;
    $y :=$ SHORTDOWN_TEST$(x, error)$ rundet x zur nächstkleineren real–Zahl y;
    error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-12}$;

14. **function** TRUNC (b: Longreal): integer;
    n:=TRUNC(x) liefert die integer–Zahl n, die durch Abschneiden der Nachkommastellen von x entsteht.
    integer–Bereich: $-2147483648 \leq \mathrm{n} \leq +2147483647$;

15. **function** TRUNC (b: Longreal): Longreal;
    y:=TRUNC(x) liefert die ganze Longreal–Zahl y, die durch Abschneiden der Nachkommastellen von x entsteht.
    y–Bereich: $-2147483648 \leq \mathrm{y} \leq +2147483647$;

16. **function** SIGN (b: Longreal): integer;
    y:=SIGN(x) liefert das Vorzeichen von x; $\mathrm{y} \in \{-1, 0, +1\}$;

17. **function** MANT (b: Longreal): Longreal;
    m:=MANT(x) liefert die vorzeichenbehaftete Mantisse von x;
    es gilt: $\mathrm{m} = 0$ oder $0.1 \leq |\mathrm{m}| < 1$;

18. **function** EXPO (b: Longreal): integer;
    ex:=EXPO(x) liefert den Zehnerexponenten ex bzgl. der Darstellung: $\mathrm{x} = \mathrm{m} \cdot 10^{\mathrm{ex}}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: $\mathrm{m} = 0$ oder $0.1 \leq |\mathrm{m}| < 1$;
    Für $\mathrm{x} \neq 0$ gilt: $-530 \leq \mathrm{ex} \leq +512$;
    Im Fall $\mathrm{x} = 0$ gilt: $\mathrm{ex} = -2147483647$;

19. **function** EXPO_0 (b: Longreal): integer;
    ex:=EXPO_0(x) liefert den Zehnerexponenten ex bzgl. der Darstellung: $x = m \cdot 10^{ex}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: $m = 0$ oder $0.1 \le |m| < 1$;
    Es gilt: $-530 \le ex \le +512$;
    Im Fall $x = 0$ gilt: $ex = 0$, im Gegensatz zur EXPO–Funktion!

20. **procedure** MANT_EXPO_PRODUCT(F1,F2: Longreal; var MP: Longreal; var exp: integer);
    MANT_EXPO_PRODUCT liefert zu den Longreal–Zahlen F1,F2 die zur nächsten Longreal–Zahl gerundete Mantisse MP des exakten Produkts $F1 \cdot F2$ und den zugehörigen Zehnerexponenten exp.
    Im Fall $F1 \cdot F2 = 0$ gilt $MP = 0$ und $exp = -maxint = -2147483647$, und im Fall $F1 \cdot F2 \neq 0$ gilt: $0.1 \le |MP| < 1$.

21. **function** COMP (m: Longreal; ex: integer): Longreal;
    y:=COMP(m,ex) liefert die Longreal–Zahl y, zusammengesetzt aus der Mantisse m und dem Zehnerexponenten ex bzgl. der Darstellung: $x = m \cdot 10^{ex}$, wobei die Mantisse m folgende Bedingungen erfüllen muß: $m = 0$ oder $0.1 \le |m| < 1$;
    Es muß gelten: $-530 \le ex \le +512$, und im Bereich $-530 \le ex \le -511$ muß die Mantisse m, wie auf S. 23 beschrieben, zusätzlich eingeschränkt sein!
    Beispiele:
    y:=COMP(Long(0.1), −531) $\longrightarrow$ $y = 0$; „Exponent to small“–Meldung; kein Programmabbruch!
    y:=COMP(Long(0.123 . . . 23) + Long(4.5678901 E − 14), −511) $\longrightarrow$ $y = 1.23\ldots 90\mathbf{0}E - 512$; „Mantissa bits lost...“–Meldung; kein Programmabbruch!
    y:=COMP(Long(0.1), +513) $\longrightarrow$ „Overflow–Meldung“ mit Programmabbruch.

22. **function** PRED (b: Longreal): Longreal;
    y:=PRED(x) liefert bzgl. x die nächstkleinere MaschinenZahl $y < x$

23. **function** SUCC (b: Longreal): Longreal;
    y:=SUCC(x) liefert bzgl. x die nächstgrößere MaschinenZahl $y > x$

24. **function** LVAL (s: string): Longreal;
    y:=LVAL(s) wandelt den String s in eine Longreal–Zahl y. Der String s muß eine Zeichenfolge enthalten, die eine Longreal–Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird durch Runden zur nächsten Longreal–Zahl berücksichtigt.

25. **function** LVAL (s: string; round: integer): Longreal;
    y:=LVAL(s,round) wandelt den String s in eine Longreal–Zahl y. Der String s muß eine Zeichenfolge enthalten, die eine Longreal–Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird durch Runden gemäß round berücksichtigt:
$$\text{round} = \left\{ \begin{array}{rl} -1 & \text{nach unten gerundet} \\ 0 & \text{nächstliegend gerundet} \\ +1 & \text{nach oben gerundet} \end{array} \right\}$$

26. **function** LVAL (s: string; round: integer; var rest: string): Longreal;
    Wie obige Funktion; ein eventuell vorhandener Reststring wird an rest zurückgegeben.

27. **function** IMAGE (s: Longreal): string;
    s:=IMAGE(y) wandelt die Longreal–Zahl y in einen String s entsprechend der Standardausgabe von Longreal–Werten.
    Beispiel:
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-7))$ liefert: $\mathrm{s} = -2.4\mathrm{E}-007\,;$

28. **function** IMAGE (s: Longreal; width: integer): string;
    s:=IMAGE(y,w) wandelt die Longreal–Zahl y in einen String s aus mindestens w Zeichen entsprechend der Ausgabe von Longreal–Werten, eventuell mit Auffüllen von Nullen am Mantissenende.
    Beispiele:
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-7), 0) \longrightarrow \mathrm{s} = -2.4\mathrm{E}-007\,;$
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-7), 11) \longrightarrow \mathrm{s} = -2.350\mathrm{E}-007\,;$

29. **function** IMAGE (s: Longreal; width, fracs: integer): string;
    s:=IMAGE(y,w,f) wandelt die Longreal–Zahl y in einen String s aus wenigstens w Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) entsprechend der Ausgabe von Longreal–Werten in Festkommadarstellung mit f Nachkommastellen, wobei Nullen angehängt werden, wenn f unnötig groß ist.
    Beispiele:
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-3), 12, 7) \longrightarrow \mathrm{s} = \sqcup\sqcup -0.0023500\,;$
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-3), 7, 4) \longrightarrow \mathrm{s} = -0.0024\,;$
    $\mathrm{s} := \mathrm{IMAGE}(\mathrm{LONG}(-2.35\,,\mathrm{E}-3), 3, 4) \longrightarrow \mathrm{s} = -0.0024\,;$

30. **function** IMAGE (s: Longreal; width, fracs, round: integer): string;
    s:=IMAGE(y,w,f,round) wandelt die Longreal–Zahl y in einen String s aus wenigstens w Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) entsprechend der Ausgabe von Longreal–Werten in Festkommadarstellung mit f Nachkommastellen, wobei Nullen angehängt werden, wenn f unnötig groß ist. Bei zu kleinem f wird entsprechend round gerundet, vergl die entsprechende Funktion LVAL.

Beispiele:
s := IMAGE(LONG(-2.35, E $-$ 3), 7, 4, 0) $\longrightarrow$ s = -0.0024;
s := IMAGE(LONG(-2.35, E $-$ 3), 3, 4, +1) $\longrightarrow$ s = -0.0023;
s := IMAGE(LONG(-2.35, E $-$ 3), 7, 4, -1) $\longrightarrow$ s = -0.0024;

31. **function** ENTIRE (x: Longreal; var gerade: boolean): boolean;
bl:=ENTIRE(x, gerade) liefert an bl den Wert true, wenn die Longreal–Zahl x eine **ganze** Zahl ist, sonst erhält bl den Wert false. Falls x ganzzahlig ist, erhält gerade den Wert true (false), wenn x zusätzlich gerade (ungerade) ist.
Ist x nicht ganzzahlig, so bleibt die Variable gerade bedeutungslos.

32. **function** EM_GANZ (x: Longreal): Longreal;
y:=EM_GANZ(x) berechnet zur Longrealzahl x die zur Mantisse von x gehörige **ganze** Zahl y.
Beispiele:
$$\begin{aligned} x &= 12.3456789012345678901E-55 \quad \longrightarrow \\ y &= 1.23456789012345678901E+20\,; \end{aligned}$$
x = -12.34 $\longrightarrow$ y = -1234.0

33. **procedure** ZERL (x: Longreal; var x1, x2: real);
ZERL (x, x1, x2) zerlegt die Longrealzahl x in zwei Summanden x1,x2 vom Typ real, mit: x = x1 + x2;
Im Fall $|x| > 1.999999999999E+256$ erfolgt eine Fehlermeldung mit Programmabbruch.
Im Fall $|x| < 1.0E-267$ gilt: x1 = x2 = 0 **ohne Fehlermeldung!!** Dies ist für viele Anwendungen nützlich!

Beispiele:

x = 1.999999999999E+256 $\longrightarrow$
x1 = 9.999999999999E+255; x2 = 9.999999999991E+255;
x = 0.9999E$-$267 $\longrightarrow$ x1 = x2 = 0 **ohne Fehlermeldung!!**
x = 1.1E$-$267 $\longrightarrow$ Fehlermeldung wegen zu breiter Mantisse von x, vergl. dazu die Beispiele auf Seite 12.
x = 1.0E$-$267 $\longrightarrow$ x1 = 1.0E$-$267, x2 = 0;

34. **priority M_10HOCH** = $*$;
operator M_10HOCH (x: Longreal; k: integer) lrint: Longreal;
y := x M_10HOCH k berechnet zur Longreal–Zahl x das Produkt $y = x \cdot 10^k$, $k \in \{0, \pm1, \pm2, \dots\}$;
Im Fall $10^{-531} \le |x \cdot 10^k| < 10^{-511}$ wird ohne Fehlermeldung zur nächsten Rasterzahl gerundet, wenn das Ergebnis wegen einer zu langen Mantisse von x nicht mehr exakt dargestellt werden kann.
Im Fall $0 < |x \cdot 10^k| < 10^{-531}$ wird das Ergebnis auf Null gesetzt.

3.2.4 Mathematische Konstanten

Es existieren 29 mathematische Konstanten im Longreal–Format, die durch Aufruf entsprechender Funktionsnamen (ohne Argument) zur Verfügung stehen:

Konstanten im Longreal – Format				
Name	**Wert**	**Funktionswert**	**Rundung**	
MAXLONGREAL		$+9.9\ldots9\cdot10^{+511}$		
MINLONGREAL		$+1.0\ldots0\cdot10^{-511}$		
MIN_LONGREAL		$+1.0\ldots0\cdot10^{-531}$		
PI_DOWN	π	$+3.1415\ldots23846$	$*$	PI_DOWN $<\pi$
PI_UP	π	$+3.1415\ldots23847$	$\uparrow$	PI_UP $>\pi$
PI2	2π	$+6.2831\ldots47693$	$*$	PI2 $>2\pi$
PI_HALF	$\pi/2$	$+1.5707\ldots61923$	$*$	PI_HALF $<\pi/2$
PI_HALF1	$\pi/2$	$+1.5707\ldots61924$	$\uparrow$	PI_HALF1 $>\pi/2$
PI_DIV4	$\pi/4$	$+0.7853\ldots09615$	$*$	PI_DIV4 $<\pi/4$
R_PI	$1/\pi$	$+0.3183\ldots71538$	$*$	R_PI $>1/\pi$
R_2PI	$1/(2\pi)$	$+0.1591\ldots35769$	$*$	R_2PI $>1/(2\pi)$
R_SQRTPI2	$2/\sqrt{\pi}$	$+1.1283\ldots57390$	$*$	R_SQRTPI2 $>2/\sqrt{\pi}$
SQRT_PI	$\sqrt{\pi}$	$+1.7724\ldots02730$	$*$	SQRT_PI $>\sqrt{\pi}$
R_SQRTPI	$1/\sqrt{\pi}$	$+0.5641\ldots86948$	$*$	R_SQRTPI $<1/\sqrt{\pi}$
SQRT_PI_D2	$\sqrt{\pi}/2$	$+0.8862\ldots13649$	$*$	SQRT_PI_D2 $<\sqrt{\pi}/2$
LN_PI	$\ln(\pi)$	$+1.1447\ldots17414$	$*$	LN_PI $<\ln(\pi)$
EUL	e	$+2.7182\ldots23536$	$*$	EUL $<e$
LN_10	$\ln(10)$	$+2.3025\ldots68402$	$*$	LN_10 $>\ln(10)$
R_LN_10	$1/\ln(10)$	$+0.4342\ldots27651$	$*$	R_LN_10 $<1/\ln(10)$
LN_2	$\ln(2)$	$+0.6931\ldots09417$	$*$	LN_2 $<\ln(2)$
R_LN_2	$1/\ln(2)$	$+1.4426\ldots40736$	$*$	R_LN_2 $>1/\ln(2)$
GAMMA_C	C	$+0.5772\ldots60607$	$*$	GAMMA_C $>C$
LN_GAM_C	$\ln(C)$	$-0.5495\ldots22338$	$*$	LN_GAM_C $>\ln(C)$
SQRT2	$\sqrt{2}$	$+1.4142\ldots04880$	$*$	SQRT2 $<\sqrt{2}$
SQRT3	$\sqrt{3}$	$+1.7320\ldots29353$	$*$	SQRT3 $>\sqrt{3}$
SQRT5	$\sqrt{5}$	$+2.2360\ldots69641$	$*$	SQRT5 $>\sqrt{5}$
SQRT10	$\sqrt{10}$	$+3.1622\ldots33200$	$*$	SQRT10 $>\sqrt{10}$
R_SQRT2	$1/\sqrt{2}$	$+0.7071\ldots24401$	$*$	R_SQRT2 $>1/\sqrt{2}$
R_SQRT2PI	$1/\sqrt{2\pi}$	$+0.3989\ldots77940$	$*$	R_SQRT2PI $>1/\sqrt{2\pi}$

$*$: Rundung zur nächsten, $\uparrow$: Rundung zur nächstgrößeren Longreal-Zahl.

3.2.5 Wertzuweisungen

Durch Überladen des Zuweisungsoperators := sind folgende Wertzuweisungen möglich:

Longrealvariable := integer–Ausdruck;
Longrealvariable := real–Ausdruck;
Longrealvariable := Longreal–Ausdruck;

Realvariable := integer–Ausdruck;
Realvariable := real–Ausdruck;
Realvariable := Longreal–Ausdruck;

Wird ein Longreal–Ausdruck in einer real–Variablen gespeichert, so wird im Bedarfsfall zur nächsten real–Zahl gerundet.

Das folgende Beispiel zeigt die PASCAL–interne Realisierung der Wertzuweisung eines real–Ausdrucks an eine Longrealvariable; dabei ist die Benutzung von **var** vor dem linken Longreal–Operanden zwingend vorgeschrieben!

operator := (**var** x: Longreal; r: real);
begin x := LONG(r) **end.**

3.2.6 Hornerschema

Gegeben sei das Polynom:

$$\mathrm{P}_{10}(x) = b[0] + \ldots + b[5] \cdot x^5 + a[6] \cdot x^6 + \ldots + a[10] \cdot x^{10},$$

das für das Longreal–Argument x nach dem Hornerschema auszuwerten ist. Zur Laufzeitverkürzung sollen die ersten 4 Schritte im groben real–Format durchgeführt werden. Im fünften, gemischten Hornerschritt wird die Multiplikation im real–Format und die Addition im Longrealformat ausgeführt. Die restlichen Hornerschritte werden dann vollständig im Longreal–Format berechnet.

Im folgenden PASCAL–XSC–Programm **HORNER** benutzen wir die im Sprachkern von **PASCAL-XSC** definierten Datentypen:

type rvector = dynamic array[*] of real;
type Lrvector = dynamic array[*] of Longreal;

Die Polynomauswertung nach dem Hornerschema erfolgt mit der Funktion:

function **HORNER_L** (a: rvector; b: Lrvector; x: Longreal): Longreal;

die vom Modul STDMOD exportiert wird.

```
program HORNER(input,output);
var  a : rvector[6..10];
     b : Lrvector[0..5];
     null : rvector[0..0];
     x, y, p10 : Longreal;
begin
     a[6] := 1.2;   . . .  a[10] := 5.3E−6;
     b[0] := LONG(1) + LONG(1.2E−13);   . . .
     b[5] := LONG(4) + LONG(3.15E−13);
     x := 0.1;
     p10 := HORNER_L (a,b,x);
     y := HORNER_L (null,b,x);
end.
```

Anmerkungen:

1. Bei 4 groben Schritten benötigt man also $4 + 1 = 5$ Komponenten a[k].
2. Ist N der Polynomgrad und $g \geq 1$ die Zahl der groben Schritte im real–Format, so muß gelten: $1 \leq g \leq N - 2$ mit $N \geq 3$;
3. p10 ist eine Maschinennäherung für den exakten Wert $P_{10}(0.1)$;
4. y := HORNER_L (null,b,x); berechnet eine Maschinennäherung für das Polynom 5-ter Ordnung, dessen Longreal–Koeffizienten im b–Feld gespeichert sind, wenn **alle Hornerschritte hochgenau** im Longrealformat ausgeführt werden.

3.2.7 Suche eines Teilintervalls

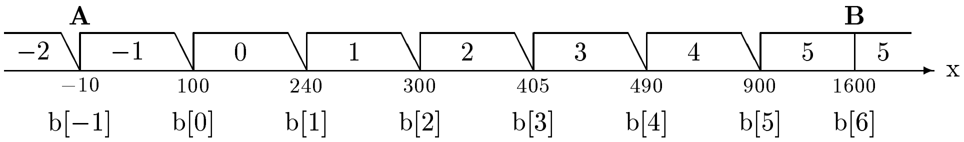


Abbildung 3.1: [A,B] *mit numerierten, halboffenen Teilintervallen.*

Für die Auswertung einer reellwertigen Funktion über einem Intervall $[A, B]$ ist es oft sinnvoll, $[A, B]$ mittels eines b–Feldes vom Typ rvector (vergl. Seite 34) in Teilintervalle aufzuteilen und für jedes Teilintervall einen eigenen Algorithmus anzugeben. Dadurch entsteht die Aufgabe, zu gegebenem Longrealargument $x \in [A, B]$ die entsprechend richtige Teilintervallnummer n zu bestimmen. In der Abb. 3.1 besitzt z.B. das halboffene Teilintervall $[300; 405[$ die Nummer 2 entsprechend dem Indexwert 2 des linken Randpunktes $b[2] = 300$. Die gesuchte Teilintervallnummer n wird bestimmt mit Hilfe der Funktion:

function **INT_NR** (b: rvector; x: Longreal): integer;

die vom Modul STDMOD exportiert wird. Das folgende Programm NUMMER zeigt eine Anwendung gemäß Abb. 3.1

```
program NUMMER(input,output);
var   b : rvector[-1..6];
      x : Longreal;
      n : integer;
begin
      b[−1] := −10;   ...   b[6] := 1600;
      x := −10.1;
      n := INT_NR (b,x);
end.
```

Ergebnisse:

n := INT_NR (b,LONG(1240.3))	liefert:	n = 5;
n := INT_NR (b,LONG(1601))	liefert:	n = 5;
n := INT_NR (b,LONG(1600))	liefert:	n = 5;
n := INT_NR (b,LONG(−10.1))	liefert:	n = −2;
n := INT_NR (b,LONG(−10))	liefert:	n = −1;
n := INT_NR (b,LONG(240))	liefert:	n = 1;

Anmerkungen:

1. Es muß gelten: A = b[−1] und B = b[6];
2. Durch geeignete Indexgrenzen des b–Feldes kann man sich einen entsprechenden Bereich für die Teilintervallnummern aussuchen!

3.2.8 Einschließung eines exakten Funktionswertes

Für einen reellwertigen Funktionswert $f(x)$ sei eine Maschinennäherung $\widetilde{f}(x)$ vom Typ Longreal berechnet. Der relative Fehler ε_f ist definiert durch:

$$\varepsilon_f := \frac{\widetilde{f}(x) - f(x)}{f(x)}\,; \qquad f(x) \neq 0\,; \qquad |\varepsilon_f| \le \varepsilon(f)\,;$$

Die Fehlerschranke $\varepsilon(f)$ sei für alle $x \in [a,b]$ bekannt. Gesucht sind eine garantierte Unterschranke US und eine Oberschranke OS für den exakten Funktionswert $f(x)$:

$$\text{US} \le f(x) \le \text{OS}\,; \qquad \text{US,OS } = ?$$

Für $\widetilde{f}(x) > 0$ und $\varepsilon(f) < 0.5 \cdot 10^{-3}$ gilt:

$$f(x) = \frac{\widetilde{f}(x)}{1+\varepsilon_f} \le \frac{\widetilde{f}(x)}{1-\varepsilon(f)} < \widetilde{f}(x) \cdot [1 + 1.001 \cdot \varepsilon(f)]$$

und mit $\alpha := 1.001 {*}{>}\, \varepsilon(f)$ erhält man: $f(x) < \widetilde{f}(x) \cdot [1+\alpha]$. Setzt man $\widetilde{f}(x) = m \cdot 10^{ex}$, so gilt: $m := \text{MANT}(\widetilde{f}(x))\,;$ $ex := \text{EXPO}(\widetilde{f}(x))$. Nach Seite 29 folgt: $0.1 \le m < 1\,;$ $\Longrightarrow$ $m \cdot [1+\alpha] = m + \alpha \cdot m < m + \alpha \le m {+}{>}\, \alpha$ $\Longrightarrow$

$$f(x) < 10^{ex} \cdot [m {+}{>}\, \alpha] =: \text{OS}$$

Die Oberschranke OS kann daher **ohne zeitaufwendige Multiplikation** berechnet werden, wenn man $[m{+}{>}\ \alpha]$ und 10^{ex} mit dem schnellen Operator M_10HOCH verknüpft!

Die Vermeidung der Multiplikation im Ausdruck $\widetilde{f}(x) \cdot [1+\alpha]$ ergibt jedoch im ungünstigsten Fall eine Überschätzung der Oberschranke als wäre $\varepsilon(f)$ verzehnfacht worden. Im Fall $\widetilde{f}(x) < 0$ können die Unter– und Oberschranken US,OS ganz analog berechnet werden. Mit Hilfe der Funktion

function **BOUND** (y: Longreal; alpha: real): Longreal ,

die vom Modul STDMOD exportiert wird, lassen sich US, OS bestimmen.
Sei y := $\widetilde{f}(x)$ und $\alpha := 1.001{*}{>}\ \varepsilon(f)$, dann liefert:

OS := BOUND (y,+alpha)	eine Oberschranke OS und
US := BOUND (y,−alpha)	eine Unterschranke US für $f(x)$.

Anmerkungen:

- Im Fall $\widetilde{f}(x) = 0$ gilt: US $= -2 \cdot 10^{-531}$; OS $= +2 \cdot 10^{-531}$; dadurch wird berücksichtigt, daß $\widetilde{f}(x) = 0$ durch Underflow entstanden sein kann!
- LONG sollte nicht mit alpha = 0 aufgerufen werden.

3.3 Ein-/Ausgabeprozeduren

Mit Hilfe der bekannten Ein-/Ausgabeprozeduren *read, readln, write, writeln* können Dateien (Filevariable) in bekannter Weise zur Ein– bzw. Ausgabe benutzt werden. Genaue Informationen findet man in: *PASCAL–XSC, Sprachbeschreibung mit Beispielen.*

Die Standardausgabe bei Verwendung der Prozedur *write* erfolgt für die Typen *integer, real, Longreal, char* und *boolean* entsprechend der Formatspezifikationstabelle 3.4.

Typ	*integer*	*real*	*Longreal*	*char*	*boolean*
Format	: 11	20 : 0 : 0	28 : 0 : 0	: 1	: 5

Tabelle 3.4: Standardausgabeformat für write.

Anmerkungen:

1. Mit *read, readln, write, writeln* ergeben sich **keine Konversionsfehler** wie sie z.B. bei einem Binär–System unvermeidlich sind!
2. Wird durch *read(x)* vom Textfile ein Wert gelesen, der betragsmäßig kleiner als $10^{-267}(10^{-531})$ ist, so erhält die real–(Longreal)Variable den Wert 0.

Besitzt die Mantisse des Textfile–Wertes mehr Ziffern, als die interne Darstellung zuläßt, so wird zur nächstgelegenen Maschinenzahl gerundet.

3. Wird durch *read(x:r)* vom Textfile ein Wert gelesen, dessen Mantisse mehr Ziffern enthält, als die interne Darstellung zuläßt, so wird entsprechend r gerundet:

$$\mathrm{r} = \left\{ \begin{array}{rl} -1 & \text{Rundung zur nächstkleineren Zahl} \\ 0 & \text{Rundung zur nächsten Rasterzahl} \\ +1 & \text{Rundung zur nächstgrößeren Zahl} \end{array} \right\}$$

x kann vom Typ *real* oder *Longreal* sein!

3.4 Exakte Auswertung von Ausdrücken

Um das Skalarprodukt

$$s := \sum_{k=1}^{10} a[k] * b[k]$$

mit nur einer einzigen Rundung berechnen zu können, gibt es in PASCAL–XSC das Konzept des **dotprecision–Ausdrucks** oder auch **Lattenkreuzausdrucks**. In der BCD–Version stehen solche **#–Ausdrücke** ebenfalls in vollem Umfang zur Verfügung; allerdings dürfen die $a[k], b[k]$ in #–Ausdrücken **nicht** vom Datentyp Longreal gewählt werden!

Mit Hilfe des Moduls STDMOD werden dem Anwender jedoch zusätzlich folgende Prozeduren und Operatoren zur Verfügung gestellt, mit denen unter gewissen Einschränkungen auch Longreal–Zahlen und exakte Produkte aus Longreal–Zahlen zu einer dotprecision–Variablen addiert (subtrahiert) werden können. Zusätzlich läßt sich der Akkumulator ins real– und Longreal–Format auslesen, wobei die Bereiche (3.1) und (3.2) entsprechend berücksichtigt werden können:

- operator + (d: dotprecision; a: Longreal) s : dotprecision;
  Zur dotprecision–Variablen d wird ein Longreal–Ausdruck addiert.
- operator + (a: Longreal; d: dotprecision) s : dotprecision;
  Zu einem Longreal–Ausdruck wird die dotprecision–Variable d addiert.
- operator − (d: dotprecision; a: Longreal) s : dotprecision;
  Von der dotprecision–Variablen d wird ein Longreal–Ausdruck subtrahiert.
- operator − (a: Longreal; d: dotprecision) s : dotprecision;
  Von einem Longreal–Ausdruck wird die dotprecision–Variable d subtrahiert.
- procedure **ADDACCU** (var d: dotprecision; a,b: Longreal);
  Zur dotprecision–Variablen d wird das **exakte** Longreal–Produkt $a \cdot b$ addiert.
- procedure **SUBACCU** (var d: dotprecision; a,b: Longreal);
  Von der dotprecision–Variablen d wird das **exakte** Longreal–Produkt $a \cdot b$ subtrahiert.

- procedure **ADDNACCU** (var d: dotprecision; a: Lrvector);
  Zur dotprecision–Variablen d werden alle Longrealzahlen des Feldes a vom Typ Lrvector (vergl. Seite 34) addiert.

- procedure **SUBNACCU** (var d: dotprecision; a: Lrvector);
  Von der dotprecision–Variablen d werden alle Longrealzahlen des Feldes a vom Typ Lrvector (vergl. Seite 34) subtrahiert.

- procedure **PADDNACCU** (var d: dotprecision; a,b: Lrvector);
  Zur dotprecision–Variablen d wird das aus den Feldern a,b gebildete **exakte** Skalarprodukt addiert.

- procedure **PSUBNACCU** (var d: dotprecision; a,b: Lrvector);
  Von der dotprecision–Variablen d wird das aus den Feldern a,b gebildete **exakte** Skalarprodukt subtrahiert.

- procedure **Akku_times_10power_k** (var d: dotprecision; k: integer);
  Multipliziert den Akkuinhalt mit $10^k; k = 0, 1, 2, \ldots$; negative k–Werte ändern den Akkuinhalt nicht.

- function **LONGNEXT** (var Akku: dotprecision): Longreal;
  y:=LONGNEXT(Akku) liest den Akku–Inhalt in die nächste Longreal–Zahl y;

- function **LONGDOWN** (var Akku: dotprecision): Longreal;
  y:=LONGDOWN(Akku) liest den Akku–Inhalt in die nächstkleinere Longreal–Zahl y;

- function **LONGUP** (var Akku: dotprecision): Longreal;
  y:=LONGUP(Akku) liest den Akkumulator–Inhalt in die nächstgrößere Longreal–Zahl y;

- function **LONGNEXT_TEST** (var Akku: dotprecision;
  var error: boolean): Longreal;
  Liest den Akkumulator–Inhalt in die nächste Longreal–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $> 0.5 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).

- function **LONGUP_TEST** (var Akku: dotprecision;
  var error: boolean): Longreal;
  Liest den Akkumulator–Inhalt in die nächstgrößere Longreal–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).

- function **LONGDOWN_TEST** (var Akku: dotprecision;
  var error: boolean): Longreal;
  Liest den Akkumulator–Inhalt in die nächstkleinere Longreal–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).

- function **REALNEXT_TEST** (var Akku: dotprecision;
  var error: boolean): real;
  Liest den Akku–Inhalt in die nächste real–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $> 0.5 \cdot 10^{-12}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.1)).
- function **REALUP_TEST** (var Akku: dotprecision;
  var error: boolean): real;
  Liest den Akku–Inhalt in die nächstgrößere real–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-12}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.1)).
- function **REALDOWN_TEST** (var Akku: dotprecision;
  var error: boolean): real;
  Liest den Akku–Inhalt in die nächstkleinere real–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-12}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.1)).
- function **SIGN** (Akku: dotprecision) : integer;
  Liefert das Vorzeichen des Akkumulators.

Da der lange Akkumulator ursprünglich nur für real–Zahlen konzipiert wurde, kann im Gegensatz zu einer Longreal–Zahl a ein exaktes Produkt $a \cdot b$ aus den Longreal–Zahlen a, b nur unter bestimmten Einschränkungen in den Akku addiert werden.

Schreibt man z.B. die Longreal–Zahl a in der normalisierten Form

$$a = ma \cdot 10^{exa}\,, \quad \text{mit: } ma = 0 \;\text{ oder } 0.1 \leq |ma| < 1,$$

so lautet die 1. Einschränkung:

$$\boxed{exa + exb \leq 512}$$

andernfalls erfogt eine Fehlermeldung mit Programmabbruch.

Bedeutet z.B. Za die **Maximalzahl** der möglichen, **von Null verschiedenen** Mantissenziffern einer Longreal–Zahl a, so gelten für Za z.B. die Werte der nachfolgenden Tabelle :

a	$ma \cdot 10^{-530}$	$ma \cdot 10^{-529}$	$ma \cdot 10^{-511}$	$ma \cdot 10^{-510}$	$ma \cdot 10^{-509}$
Za	1	2	20	21	21

Tabelle 3.5: Za =Mantissenlänge einer Longreal–Zahl a

und die zweite Einschränkung lautet:

$$\boxed{exa + exb - Za - Zb \geq -552}$$

andernfalls erfolgt eine Fehlermeldung mit Programmabbruch.
Die zweite Einschränkung läßt sich auch wie folgt formulieren:

> Für **beliebige**, zulässige Mantissen ma, mb darf das entsprechende exakte Produkt $a \cdot b$ zweier Longrealzahlen in Festkommadarstellung höchstens bis zur 552.–ten Nachkommastelle reichen.

Da Verstöße gegen die beiden Einschränkungen durch Programmabbruch geahndet werden, sind Rechnerergebnisse mit den oben genannten Prozeduren und Operatoren absolut zuverlässig und ermöglichen z.B. den Aufbau einer hochgenauen, komplexen Longreal–Arithmetik.

> Im Gegensatz zur Multiplikation sind **Addition** und **Subtraktion** von Longreal–Zahlen in den Akkumulator **ohne Einschränkungen** möglich!

Beispiele:

Vereinbarungen: var a,b : Lrvector[1..2]; d : dotprecision;
Die Ergebnisse in Tabelle 3.7 entstanden durch die Anweisungen:

d := #(0); PADDNACCU (d,a,b);

Nr.	**a[1]**	**b[1]**	**a[2]**	**b[2]**
1	1E−1	1E−531	0	0
2	1E−2	1E−531	0	0
3	1E−256	1E−256	0	0
4	1E−256	1E−257	0	0
5	1.234..8901E−256	a[1]	−1.234..8902E−256	1.234..8900E−256
6	1E+510	1	0	0
7	9.99..99E+510	9.99..99	0	0
8	1E+511	1	0	0
9	9.99..99E+510	9.99..99	1E+491	1
10	9.99..99E+510	9.99..99	2E+491	1

Tabelle 3.6: Komponenten vom Typ Longreal.

Nr.	$\mathbf{d := \#(0);\quad paddnaccu(d,a,b);} \Longleftrightarrow \mathbf{d = 0 + a[1]\cdot b[1] + a[2]\cdot b[2];}$
1	1E−532 liefert Underflow beim Auslesen des Akkus.
2	Exponent range restricted ... ; Programmabbruch.
3	1E−512
4	Exponent range restricted ... ; Programmabbruch.
5	1E−552; kleinstmögliche, positive Zahl im Akkumulator.
6	1E+510
7	9.99...998 00...001E+511 < 1E+512
8	Invalid Operation... ; Programmabbruch.
9	9.99...999 00...001E+511 < 1E+512
10	1.000...(41Ziffern 0)...0001E+512 > 1E+512

Tabelle 3.7: Akkumulator–Inhalte.

3.4.1 Operatoren für dotprecision–Ausdrücke

Da ein #-Konzept für real- **und** Longreal–Ausdrücke nicht zur Verfügung steht, soll dieses mit Hilfe geeigneter Operatoren simuliert werden.

Um z.B. das **exakte** Produkt $a \cdot b$ der Longreal–Zahlen $a = b = 1.23456789012345678901 \cdot 10^{250}$ in die dotprecision–Variable Akku zu addieren, darf in der (erlaubten!) Anweisung

$$Akku := Akku + a * b\,;$$

der $*$ Operator **nicht** benutzt werden, da dieser das exakte Produkt $a \cdot b$ **zuerst** ins Longrealformat rundet und **danach** das **gerundete** Longreal–Produkt in den Akku addiert! Will man daher das Problem ohne die Prozedur ADDACCU übersichtlicher mit einem **Operator** lösen, so muß man für den Multiplikationsoperator mit dem Ergebnistyp dotprecision einen **neuen** Namen wählen! In nachfolgender Tabelle 3.8 sind die entsprechenden neuen Operatornamen zusammengestellt:

Operation	Operator–Name	Ergebnistyp
Addition	**plus**	dotprecision
Subtraktion	**minus**	dotprecision
Multiplikation	**times**	dotprecision

Tabelle 3.8: Operatornamen mit Ergebnistyp dotprecision

In Tabelle 3.9 sind für die Operatoren $\mathbf{plus, minus, times,} *$ alle erlaubten Operanden–Typen angegeben.

rechter Operand	integer	real	Longreal	dotprecision
linker Operand				
integer real	times plus, minus	times plus, minus	times plus, minus	$*$ plus, minus
Longreal	times plus, minus	times plus, minus	times plus, minus	$+, -, *$ plus, minus
dotprecision	$*$ plus, minus	$*$ plus, minus	$+, -, *$ plus, minus	$*$ plus, minus

Tabelle 3.9: Operatoren vom Ergebnistyp dotprecision

1. Beispiel:
Mit den Longreal–Zahlen
$a = b = 1.234567890123456$; $c = 1.234567890123457$; $d = 1.234567890123455$
liefern die Anweisungen:

```
yn := LONGNEXT ( a times b minus c times d );
yd := LONGDOWN ( a times b minus c times d );
yu := LONGUP ( a times b minus c times d );
```

jeweils den gleichen Longrealwert $yn = yd = yu = 1 \cdot 10^{-30}$, d.h. yn liefert sogar den **exakten** Wert von $a \cdot b - c \cdot d$ (vergl. Beispiel 3. von Seite 5)

2. Beispiel:
Mit Hilfe der Operatoren $*$ und **plus** zwischen *dotprecision*- und *Longreal*-Operanden kann ein Polynom

$$\mathrm{P}(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

mit der folgenden Funktion nach dem Hornerschema **rundungsfehlerfrei** berechnet werden:

```
function HORNER_dot (var a: Lrvector; xl: Longreal): dotprecision;
var k : integer;
    y : dotprecision;
begin
    y := a[ub(a)];
    for k := ub(a)−1 downto lb(a) do
    y := y ∗ xl plus a[k];
    HORNER_dot := y
end;
```

Die Koeffizienten

$a_0 \ = -0.01;$	$a_1 \ = +0.39;$	$a_2 \ = -7.02;$	$a_3 \ = +77.22;$
$a_4 \ = -579.15;$	$a_5 \ = +3127.41;$	$a_6 \ = -12509.64;$	$a_7 \ = +37528.92;$
$a_8 \ = -84440.07;$	$a_9 \ = +140733.45;$	$a_{10} = -168880.14;$	$a_{11} = +138174.66;$
$a_{12} = -69087.33;$	$a_{13} = +15943.23;$		

realisieren das Polynom $\mathrm{P}(x) = \frac{1}{100} \cdot (3x-1)^{13}$ mit der Nullstelle $x_0 = \frac{1}{3}$;
Mit den Deklarationen:

```
var a: Lrvector[0..13];
    x1,y: Longreal;   dot: dotprecision;
```

und den Anweisungen:

```
x1   := 0.3333333333333333;   {x1 ≈ x0}
dot  := HORNER_dot (a,x1);
y    := LONGNEXT(dot);
```

erhält man mit der BCD–Version das **exakte Ergebnis:**

$$y = \mathrm{P}(x1) = -1.00\ldots00 \cdot 10^{-210};$$

Führt man ganz entsprechende Rechnungen mit der Binärversion von PASCAL–XSC durch, so erhält man:

$$\widetilde{\mathrm{P}}(\widetilde{x1}) = -1.002225955526114 \cdot 10^{-15};$$

Die enorme Abweichung von **195 Größenordnungen** vom exakten BCD–Wert ist alleine bedingt durch den Umstand, daß **vor** der eigentlichen rundungsfehlerfreien Polynomauswertung neben dem Argument x1 auch alle Polynomkoeffizienten mit Konversionsfehlern ins Binärsystem umgewandelt werden müssen.
In der Binärversion kann man bei diesem Beispiel die Konversionsfehler bei den $a[k]$ vermeiden, indem man diese Polynomkoeffizienten mit dem Faktor 100 multipliziert und damit ganzzahlig macht:

$$\mathrm{Q}(x) := 100 \cdot \mathrm{P}(x) \equiv (3x-1)^{13}$$

Der noch verbleibende Konversionsfehler beim Argument $x1$ erzeugt dann mit der Binärversion im Vergleich zum exakten BCD–Wert immer noch einen Fehler von etwa vier Größenordnungen:

Exakter BCD–Wert :	$\mathrm{Q}(x1) = -1.000000000000000 \cdot 10^{-208}$
Binär-Ergebnis :	$\widetilde{\mathrm{Q}}(\widetilde{x1}) = -4.752728915737900 \cdot 10^{-212}$

Das Beispiel zeigt, daß man trotz Vermeidung der Konversionsfehler bei den Polynomkoeffizienten (was zu vorzeitigem Overflow führen kann!) nur mit einer BCD–Version akzeptable Ergebnisse erhält, wenn man ein Polynom in der Nähe einer Nullstelle auswerten will.

Vergleichsoperatoren:
Bezüglich einer beliebigen Kombination von Operanden des Typs

integer, real, Longreal, dotprecision

existieren alle Vergleichsoperatoren: $= \quad <> \quad > \quad >= \quad < \quad <= \quad ;$

Wertzuweisungen:
An eine Variable vom Typ dotprecision sind Wertzuweisungen von Ausdrücken folgender Typen möglich:

dotprecision–Variable := integer, real, Longreal, dotprecision–Ausdruck;

3.5 Der Datentyp rrG

Zur Implementierung **hochgenauer** Standardfunktionen im 21-stelligen Longreal–Format wurden die entsprechenden Algorithmen im **rrG**–Format realisiert, das wie folgt definiert

```
Type rrG = record
           r1,r2 : real;
           G     : integer;
           end;
```

und durch das Standardmodul STDMOD zur Verfügung gestellt wird.
Durch a : rrG und

$$\text{a.r1} := 1.234567890123; \quad \text{a.r2} := 4.567890123456\text{E}-13; \quad \text{a.G} := +123456789;$$

wird eine BCD-Zahl mit mindestens 26 Mantissenstellen dargestellt, deren Exponentenbereich nur durch die integer–Zahlen begrenzt ist:

$$\text{a} = 1.23456789012345678901234 56 \cdot 10^{+123456789}$$
$$-2147483648 \leq \text{a.G} \leq +2147483647$$

Für die korrekte Durchführung der arithmetischen Grundoperationen muß ein rrG–Operand a die beiden folgenden Bedingungen erfüllen:

$$\boxed{\text{a.r1} = 0 \quad \Longrightarrow \quad \text{a.r2} = 0 \qquad \text{(Beding. I)}}$$

$$\boxed{|\text{a.r1}| > 0 \quad \Longrightarrow \quad \left|\frac{\text{a.r2}}{\text{a.r1} + \text{a.r2}}\right| < 5.0001 \cdot 10^{-13} \qquad \text{(Beding. II)}}$$

Die beiden obigen Bedingungen werden dabei automatisch realisiert, falls wie in allen praktischen Anwendungen a.r1 und a.r2 durch die folgenden Anweisungen aus dem Akkumulator ausgelesen werden:

```
a.r1 := #*( Akku );         { a: rrG;  Akku: dotprecision }
a.r2 := #*( Akku - a.r1 );
```

Bedeutet Akku den numerischen Wert des Akkumulators, so gilt mit den obigen Anweisungen zusätzlich:

$$\boxed{|\text{a.r1}| \geq 10^{-241}, \quad (\text{a.r1} + \text{a.r2}) = \text{Akku} \cdot (1 + \varepsilon) \qquad \Longrightarrow \qquad |\varepsilon| \leq 5 \cdot 10^{-26}}$$

Das zweimalige Auslesen des Akkumulators erfolgt also mit dem relativen Höchstfehler $\varepsilon(26) := 5 \cdot 10^{-26}$, falls $|\text{a.r1}| \geq 10^{-241}$. Nach den beiden obigen **PASCAL-XSC** –Anweisungen erreicht man daher mit der Abfrage

```
if (SIGN(Akku) <> 0) and (EXPO(a.r1) < TEST_26)
                then  Fehlermeldung
```

daß das zweimalige Auslesen des Akkumulators in die rrG–Komponenten a.r1 und a.r2 den relativen Fehler $5 \cdot 10^{-26}$ nicht übersteigt; TEST_26 = −240 wird vom Modul STDMOD als real-Konstante exportiert.

3.5.1 Monadische Operatoren

Für den Datentyp **rrG** stehen die monadischen Operatoren **+ −** in gewohnter Weise zur Verfügung.

3.5.2 Addition, Subtraktion

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Variablen vom Typ rrG addieren bzw. subtrahieren. Falls beide Operanden den gleichen Zehnerexponenten besitzen, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen. Es existieren auch die wichtigsten Operatoren mit gemischten Operanden.

- procedure **ADD_11** (a,b: real; var r1,r2: real);
  Es gilt: $(a{+}b) \equiv (r1{+}r2)$ mit der relativen Fehlerschranke 0, falls kein Overflow auftritt. $|r2/(r1+r2)| < 5.0001 \cdot 10^{-13}$ wird erfüllt.

- procedure **ADD_21** (a1,a2,b: real; var r1,r2: real);
  $(r1+r2) = (a1+a2+b)\cdot(1+\varepsilon)$, $|r1| \geq 10^{-241} \Longrightarrow |\varepsilon| \leq 5\cdot 10^{-26}$, falls kein Overflow auftritt. $|r2/(r1+r2)| < 5.0001 \cdot 10^{-13}$ wird erfüllt.

- procedure **ADD_22** (a1,a2,b1,b2: real; var r1,r2: real);
  $(r1+r2) = (a1+a2+b1+b2)\cdot(1+\varepsilon)$, $|r1| \geq 10^{-241} \Longrightarrow |\varepsilon| \leq 5\cdot 10^{-26}$, falls kein Overflow auftritt. $|r2/(r1+r2)| < 5.0001 \cdot 10^{-13}$ wird erfüllt.

- operator **+** (x,y: rrG) s: rrG;
  Vorauss.: $|\ x.r2/(x.r1+x.r2)\| < 5.0001\cdot 10^{-13}$
  $|\ y.r2/(y.r1+y.r2)\| < 5.0001\cdot 10^{-13}$
  $s = (x+y)\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-26}$; $|s.r2/(s.r1+s.r2)| < 5.0001\cdot 10^{-13}$

- operator **+** (x: real; y: rrG) s: rrG;
  Vorauss.: $|\ y.r2/(y.r1+y.r2)\| < 5.0001\cdot 10^{-13}$
  $s = (x+y)\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-26}$; $|s.r2/(s.r1+s.r2)| < 5.0001\cdot 10^{-13}$

- operator **+** (x: Longreal; y: rrG) s: rrG;
  Vorauss.: $|\ y.r2/(y.r1+y.r2)\| < 5.0001\cdot 10^{-13}$
  $s = (x+y)\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-26}$; $|s.r2/(s.r1+s.r2)| < 5.0001\cdot 10^{-13}$

- operator **−** (x,y: rrG) d: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | < 5.0001\cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\mathrm{d} = (\mathrm{x}-\mathrm{y})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **−** (x: real; y: rrG) d: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\mathrm{d} = (\mathrm{x}-\mathrm{y})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **−** (x: Longreal; y: rrG) d: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\mathrm{d} = (\mathrm{x}-\mathrm{y})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **−** (y: rrG; x: real) d: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\mathrm{d} = (\mathrm{y}-\mathrm{x})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **−** (y: rrG; x: Longreal) d: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\mathrm{d} = (\mathrm{y}-\mathrm{x})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **+>** (x,y: rrG) s: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | < 5.0001\cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\Longrightarrow$ $\mathrm{x}+\mathrm{y} < \mathrm{s}$; $|\mathrm{s.r2}/(\mathrm{s.r1}+\mathrm{s.r2})| < 5.0001\cdot 10^{-13}$

- operator **−>** (x,y: rrG) d: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | < 5.0001\cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\Longrightarrow$ $\mathrm{x}-\mathrm{y} < \mathrm{d}$; $|\mathrm{d.r2}/(\mathrm{d.r1}+\mathrm{d.r2})| < 5.0001\cdot 10^{-13}$

- operator **+<** (x,y: rrG) s: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | < 5.0001\cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | < 5.0001\cdot 10^{-13}$
  $\Longrightarrow$ $\mathrm{x}+\mathrm{y} > \mathrm{s}$; $|\mathrm{s.r2}/(\mathrm{s.r1}+\mathrm{s.r2})| < 5.0001\cdot 10^{-13}$

- operator **−<** (x,y: rrG) d: rrG;
  Vorauss.: $|\ x.r2/(x.r1+x.r2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $|\ y.r2/(y.r1+y.r2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $\Longrightarrow$ $x - y > d;$ $\quad |d.r2/(d.r1+d.r2)| < 5.0001\cdot 10^{-13}$

- procedure **ADD_11G** (a,b: Longreal; var s: rrG);
  $s = (a+b)\cdot(1+\varepsilon) \quad \Longrightarrow \quad |\varepsilon| \le 5\cdot 10^{-26}$, $|s.r2/(s.r1+s.r2)| < 5.0001\cdot 10^{-13}$
  Für b kann auch ein real-Parameter eingesetzt werden.

3.5.3 Multiplikation

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Variablen vom Typ rrG multiplizieren. Falls die Summe der Zehnerexponenten beider Operanden verschwindet, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen. Es existieren auch die wichtigsten Operatoren mit gemischten Operanden.

- procedure **MUL_11** (a,b: real; var r1,r2: real);
  Es gilt: $a\cdot b \equiv (r1+r2)$ mit der relativen Fehlerschranke 0, falls
  $|r1| \ge 10^{-241}$, $\quad |r2/(r1+r2)| < 5.0001\cdot 10^{-13}$ wird erfüllt.

- procedure **MUL_12** (a,b1,b2: real; var r1,r2: real);
  $(r1+r2) = a\cdot(b1+b2)\cdot(1+\varepsilon)$, $\quad |r1| \ge 10^{-241}$
  $\Longrightarrow$ $|\varepsilon| \le 5\cdot 10^{-26}$, $\quad |r2/(r1+r2)| < 5.0001\cdot 10^{-13}$ wird erfüllt.

- procedure **MUL_22** (a1,a2,b1,b2: real; var r1,r2: real);
  Vorauss.: $|\ a2/(a1+a2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $|\ b2/(b1+b2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $(r1+r2) = (a1+a2)\cdot(b1+b2)\cdot(1+\varepsilon)$, $\quad |r1| \ge 10^{-241}$
  $\Longrightarrow$ $|\varepsilon| \le 3.0003\cdot 10^{-25}$; $\quad |r2/(r1+r2)| < 5.0001\cdot 10^{-13}$

- procedure **MUL_21** (a1,a2: real; xl: Longreal; var r1,r2: real);
  Vorauss.: $|a2/(a1+a2)| < 5.0001\cdot 10^{-13}$, $\quad |xl| \ge 10^{-247}$
  $(r1+r2) = (a1+a2)\cdot xl\cdot(1+\varepsilon)$, $\quad |r1| \ge 10^{-241}$
  $\Longrightarrow$ $|\varepsilon| \le 3.0003\cdot 10^{-25}$; $\quad |r2/(r1+r2)| < 5.0001\cdot 10^{-13}$

- procedure **MUL_22S** (a1,a2,b1,b2: real; var r1,r2: real);
  Vorauss.: $|\ a2/(a1+a2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $|\ b2/(b1+b2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $(r1+r2) = (a1+a2)\cdot(b1+b2)\cdot(1+\varepsilon)$, $\quad |r1| \ge 10^{-241}$
  $\Longrightarrow$ $|\varepsilon| \le 5\cdot 10^{-26}$; $\quad |r2/(r1+r2)| < 5.0001\cdot 10^{-13}$

- operator ***** (x,y: rrG) p: rrG;
  Vorauss.: $|\ x.r2/(x.r1+x.r2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $|\ y.r2/(y.r1+y.r2)\ | \ < \ 5.0001\cdot 10^{-13}$
  $p = (x\cdot y)\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \le 5.0001\cdot 10^{-26}$; $\quad |p.r2/(p.r1+p.r2)| < 5.0001\cdot 10^{-13}$

- operator $*$ (x: real; y: rrG) p: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\mathrm{p} = (\mathrm{x} \cdot \mathrm{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001 \cdot 10^{-26}$; $|\mathrm{p.r2}/(\mathrm{p.r1}+\mathrm{p.r2})| < 5.0001 \cdot 10^{-13}$

- operator $*$ (x: Longreal; y: rrG) p: rrG;
  Vorauss.: $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\mathrm{p} = (\mathrm{x} \cdot \mathrm{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001 \cdot 10^{-26}$; $|\mathrm{p.r2}/(\mathrm{p.r1}+\mathrm{p.r2})| < 5.0001 \cdot 10^{-13}$

- operator **$*$>** (x,y: rrG) p: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ $\mathrm{x} \cdot \mathrm{y} < \mathrm{p}$; $|\mathrm{p.r2}/(\mathrm{p.r1}+\mathrm{p.r2})| < 5.0001 \cdot 10^{-13}$

- operator **$*$<** (x,y: rrG) p: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ $\mathrm{x} \cdot \mathrm{y} > \mathrm{p}$; $|\mathrm{p.r2}/(\mathrm{p.r1}+\mathrm{p.r2})| < 5.0001 \cdot 10^{-13}$

3.5.4 Division

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Quotienten vom Typ rrG berechnen. Falls die Zehnerexponenten beider rrG–Operanden übereinstimmen, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen. Es existieren auch die wichtigsten Operatoren mit gemischten Operanden.

- procedure **DIV_22** (a1,a2,b1,b2: real; var q1,q2: real);
  Vorauss.: $|\ \mathrm{a2}/(\mathrm{a1}+\mathrm{a2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $|\ \mathrm{b2}/(\mathrm{b1}+\mathrm{b2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $(\mathrm{q1}+\mathrm{q2}) = [(\mathrm{a1}+\mathrm{a2})/(\mathrm{b1}+\mathrm{b2})] \cdot (1+\varepsilon)$, $|\mathrm{q1}| \geq 10^{-241}$
  $\Longrightarrow$ $|\varepsilon| \leq 2.2504 \cdot 10^{-24}$; $|\mathrm{r2}/(\mathrm{r1}+\mathrm{r2})| < 5.0001 \cdot 10^{-13}$

- procedure **REZIP** (x: rrG; var R: rrG);
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\mathrm{R} = (1/x) \cdot (1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 2.2504 \cdot 10^{-24}$; $|\mathrm{R.r2}/(\mathrm{R.r1}+\mathrm{R.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/** (x,y: rrG) q: rrG;
  Vorauss.: $|\ \mathrm{x.r2}/(\mathrm{x.r1}+\mathrm{x.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $|\ \mathrm{y.r2}/(\mathrm{y.r1}+\mathrm{y.r2})\ | \ < \ 5.0001 \cdot 10^{-13}$
  $\mathrm{q} = (\mathrm{x}/\mathrm{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 2.2505 \cdot 10^{-24}$; $|\mathrm{q.r2}/(\mathrm{q.r1}+\mathrm{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/** (x: real; y: rrG) q: rrG;
  Vorauss.: $|\ \text{y.r2}/(\text{y.r1}+\text{y.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\text{q} = (\text{x}/\text{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 2.2505 \cdot 10^{-24}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/** (x: Longreal; y: rrG) q: rrG;
  Vorauss.: $|\ \text{y.r2}/(\text{y.r1}+\text{y.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\text{q} = (\text{x}/\text{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 2.2505 \cdot 10^{-24}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/** (x: rrG; y: real) q: rrG;
  Vorauss.: $|\ \text{x.r2}/(\text{x.r1}+\text{x.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\text{q} = (\text{x}/\text{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 2.2505 \cdot 10^{-24}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/** (x: rrG; y: Longreal) q: rrG;
  Vorauss.: $|\ \text{x.r2}/(\text{x.r1}+\text{x.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\text{q} = (\text{x}/\text{y}) \cdot (1+\varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 2.2505 \cdot 10^{-24}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/>** (x,y: rrG) q: rrG;
  Vorauss.: $|\ \text{x.r2}/(\text{x.r1}+\text{x.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ \text{y.r2}/(\text{y.r1}+\text{y.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow \quad \text{x}/\text{y} < \text{q}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

- operator **/<** (x,y: rrG) q: rrG;
  Vorauss.: $|\ \text{x.r2}/(\text{x.r1}+\text{x.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ \text{y.r2}/(\text{y.r1}+\text{y.r2})\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow \quad \text{x}/\text{y} > \text{q}; \qquad |\text{q.r2}/(\text{q.r1}+\text{q.r2})| < 5.0001 \cdot 10^{-13}$

3.5.5 Vergleiche

Mit Hilfe der folgenden Funktionen und Operatoren lassen sich Operanden vom Typ real,Longreal,rrG vergleichen:

- function **greater_21** (x1,x2,x: real): boolean;
  Vorauss.: $|\text{x2}/(\text{x1}+\text{x2})| < 5.0001 \cdot 10^{-13}, \qquad \text{x1} = 0 \Longrightarrow \text{x2} = 0;$
  greater_21 = true $\Longleftrightarrow \quad (x1+x2) > x;$

- function **less_21** (x1,x2,x: real): boolean;
  Vorauss.: $|\text{x2}/(\text{x1}+\text{x2})| < 5.0001 \cdot 10^{-13}, \qquad \text{x1} = 0 \Longrightarrow \text{x2} = 0;$
  less_21 = true $\Longleftrightarrow \quad (x1+x2) < x;$

- operator = (x,y: rrG) eq: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ eq = true $\iff$ x = y;

- operator > (x,y: rrG) gr: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ gr = true $\iff$ x > y;

- operator < (x,y: rrG) le: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ le = true $\iff$ x < y;

- operator ≥ (x,y: rrG) ge: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ ge = true $\iff$ x ≥ y;

- operator ≤ (x,y: rrG) le: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ le = true $\iff$ x ≤ y;

- operator <> (x,y: rrG) neq: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $|\ y.r2/(y.r1 + y.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ neq = true $\iff$ x <> y;

- operator = (x: rrG; a: real) eq: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ eq = true $\iff$ x = a;

- operator <> (x: rrG; a: real) neq: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ neq = true $\iff$ x <> a;

- operator > (x: rrG; a: real) gr: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ gr = true $\iff$ x > a;

- operator < (x: rrG; a: real) le: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ le = true $\iff$ x < a;

- operator $\geq$ (x: rrG; a: real) ge: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ ge = true $\Longleftrightarrow$ $x \geq a$;

- operator $\leq$ (x: rrG; a: real) le: boolean;
  Vorauss.: $|\ x.r2/(x.r1 + x.r2)\ | \quad < \quad 5.0001 \cdot 10^{-13}$
  $\Longrightarrow$ le = true $\Longleftrightarrow$ $x \leq a$;

Bei den letzten Operatoren kann a auch vom Typ **Longreal** gewählt werden, so daß Vergleiche auch zwischen rrG- und Longreal–Operanden möglich sind.

3.5.6 Transferfunktionen und Zuweisungsoperatoren

Die folgenden Funktionen und Operatoren ermöglichen die Verbindung zwischen den Datentypen **real, Longreal, rrG** :

- operator **:=** (var a: rrG; r: real);

- operator **:=** (var a: rrG; r: Longreal);

- function **LongNext** (x: rrG) : Longreal; Rundg. zur nächsten Longrealzahl

- function **LongUp** (x: rrG) : Longreal; zur nächstgrößeren Longrealzahl;

- function **LongDown** (x: rrG) : Longreal; zur nächstkleineren Longrealzahl;

- function **RealNext** (x: rrG) : real; Rundung zur nächsten real–Zahl;

- function **Trans_rrG** (x1,x2: Longreal) : rrG;
  Vorauss.: $x1 = 0 \Rightarrow x2 = 0$; $\quad x1 <> 0 \Rightarrow |x2/(x1 + x2)| < 5.0001 \cdot 10^{-21}$;
  y := Trans_rrG(x1,x2) liefert: $y = (x1 + x2) \cdot (1 + \varepsilon) \quad \longrightarrow \quad |\varepsilon| \leq 5 \cdot 10^{-26}$;

3.5.7 Hilfsfunktionen

Bei den auftretenden Funktionsparametern vom Typ **rrG** wird vorausgesetzt, daß die Bedingungen I,II von Seite 46 erfüllt sind. Dies wird immer dann der Fall sein, wenn die real–Komponenten dieser Parameter durch das auf der gleichen Seite beschriebene zweimalige Auslesen des Akkumulators ins real–Format definiert sind. Besitzen Funktionen den Ergebnistyp rrG, so erfüllen auch die Komponenten aller Funktionswerte automatisch die Bedingungen I,II von Seite 46.

- function **Expo** (x: rrG) : integer;
  Berechnet den Zehnerexponeneten analog zu Expo(x: real): integer;

- procedure **Mant_26** (x1,x2: real; var r1,r2: real);
  Berechnet die Mantisse von $(x1 + x2)$ analog zu Mant(x: real): real , falls $|x.r2/(x.r1 + x.r2)| < 5.0001 \cdot 10^{-13}$ erfüllt ist.
  Es gilt also: $|r1 + r2| = 0$ oder $0.1 \leq |r1 + r2| < 1$;

- function **Scale** (x: rrG): rrG;
  y := Scale(x) liefert folgende Ergebnisse:

  1. y = x, '=' im Sinne der Numerik;
  2. y.r1 = 0 $\Longrightarrow$ y.r2 = 0, y.G = −Maxint = −2147483647;
  3. $0.1 \leq |\text{y.r1}| < 1$, falls y.r1 <> 0; (Skalierung);
  4. $|\text{y.r2}/(\text{y.r1}+\text{y.r2})| < 5.0001 \cdot 10^{-13}$, falls y.r1 <> 0;

  Falls das Ergebnis y = x in extremen Ausnahmefällen nicht erfüllt werden kann, erfolgt eine entsprechende Fehlermeldung.

- function **Normalisiere** (x: rrG): rrG;
  y := Normalisiere(x) liefert folgende Ergebnisse:

  1. y.r1 = 0 $\Longrightarrow$ y.r2 = 0, y.G = −Maxint = −2147483647;
  2. $0.1 \leq |\text{y.r1}| < 1$, falls y.r1 <> 0; (Normalisierung);
  3. $|\text{y.r2}/(\text{y.r1}+\text{y.r2})| < 5.0001 \cdot 10^{-13}$, falls y.r1 <> 0;
  4. $\text{y} = \text{x} \cdot (1+\varepsilon) \Longrightarrow |\varepsilon| < 1.0001 \cdot 10^{-266}$;
  5. $\text{y} = \text{x} + \delta \Longrightarrow |\delta| < 10^{-267+\text{x.G}+\text{Expo(x.r1)}}$;

  Bei extrem kleinen oder großen |x|–Werten erfolgt eine Fehlermeldung, wenn y.G im integer–Bereich nicht mehr darstellbar ist, was für die Praxis jedoch keine Einschränkung bedeutet.

- function **Bound** (a: rrG; eps: real): rrG;
  a sei eine Maschinennäherung für den exakten Wert $f(x)$, und es gelte:

  1. $\text{a} = f(x) \cdot (1+\varepsilon)$, $|\varepsilon| \leq \varepsilon(f)$, epsup := 1.00001 $*\!>$ $\varepsilon(f)$;
  2. epsup $< 10^{-13}$, was keine praktische Einschränkung bedeutet;

  Mit: B1 := Bound(a,−epsup); B2 := Bound(a,+epsup) erhält man garantierte Schranken für $f(x)$: $\text{B1} < f(x) < \text{B2}$, B1,B2: rrG; a = 0 wird durch die Funktion Bound nicht auf- oder abgerundet. Bei extrem kleinen oder großen |a|–Werten erfolgt (durch die Funktion Normalisiere) eine Fehlermeldung, was für die Praxis jedoch keine Einschränkung bedeutet.

- function **Bound_toL** (a: rrG; eps: real): Longreal;
  a sei eine Maschinennäherung für den exakten Wert $f(x)$, und es gelte:

  $$\text{a} = f(x) \cdot (1+\varepsilon), \quad |\varepsilon| \leq \varepsilon(f), \quad \text{epsup} := 1.00001 *\!> \varepsilon(f);$$

  Mit: L1 := Bound_toL(a,−epsup); L2 := Bound_toL(a,+epsup) erhält man garantierte Schranken für $f(x)$: $\text{L1} < f(x) < \text{L2}$, L1,L2: Longreal;
  a = 0 liefert: $\text{L1} = -10^{-531}$, $\text{L2} = +10^{-531}$;
  $0 < \text{a} < +10^{-531}$ liefert: L1 = 0, $\text{L2} = +2 \cdot 10^{-531}$;
  $-10^{-531} < \text{a} < 0$ liefert: $\text{L1} = -2 \cdot 10^{-531}$, L2 = 0;

- function **Prod_Sum** (a,b,c,d: rrG): rrG;
  y := Prod_Sum(a,b,c,d) liefert:
  $y = (\mathrm{a} \cdot \mathrm{b} + \mathrm{c} \cdot \mathrm{d}) \cdot (1+\varepsilon), \qquad |\varepsilon| < 1.1112 \cdot 10^{-25}$;

- function **Transform** (x: rrG): rrG;
  Vorauss.: x.r1, x.r2, x.G seien beliebig;
  Durch y := Transform(x) wird x so transformiert, daß gilt:

  1. y = x;
  2. y.r1 = 0 $\Longrightarrow$ y.r2 = 0;
  3. y.r1 <> 0 $\Longrightarrow$ $|\mathrm{y.r2}/(\mathrm{y.r1} + \mathrm{y.r2})| < 5.0001 \cdot 10^{-13}$;

  Falls y = x nicht erfüllt werden kann, erfolgt eine Fehlermeldung.

- procedure **Arg_Red** (x1,x2,m: real; var r1,r2: real);
  Zu gegebenem x = (x1 + x2), mit $\pi/2 < |\mathrm{x}| < 10^{+8}$ ist (r1 + r2) eine Näherung für das reduzierte Argument xred := $x - \mathrm{m} \cdot \pi$;
  m := n/2, mit $\mathrm{n} \in \mathbb{Z}$, muß dabei so vorgegeben sein, daß gilt:
  $|\mathrm{xred}| < \pi/2 + t, \quad 0 \le t \ll 1$;
  $(\mathrm{r1} + \mathrm{r2}) = \mathrm{xred} \cdot (1+\varepsilon), \quad |\varepsilon| < 5.0002 \cdot 10^{-26}$;

- function **ENTIRE** (x: rrG; var gerade: boolean): boolean;
  bl:=ENTIRE(x, gerade) liefert an bl den Wert true, wenn die rrG–Zahl x eine **ganze** Zahl ist, sonst erhält bl den Wert false.
  Falls x ganzzahlig ist, erhält gerade den Wert true (false), wenn x zusätzlich gerade (ungerade) ist.
  Ist x nicht ganzzahlig, so bleibt die Variable gerade bedeutungslos.

3.5.8 Hornerschema

Gegeben sei das Polynom vom Grade $N = 9$:

$$\mathrm{P}_9(x) = (b1[0] + b2[0]) + (b1[1] + b2[1]) \cdot x^1 + a[2] \cdot x^2 + \ \ldots \ + a[9] \cdot x^9$$

das für das 26-stellige BCD–Argument $x = (x1 + x2)$ nach dem Hornerschema auszuwerten ist. Die real–Zahlen $x1, x2$ müssen folgende Bedingungen erfüllen:

$$x1 = 0 \quad \Longrightarrow \quad x2 = 0; \qquad\qquad x1 <> 0 \quad \Longrightarrow \quad |x2/(x1 + x2)| < 5.0001 \cdot 10^{-13}$$

Aus Laufzeitgründen werden die ersten $gr = 7$ groben Hornerschritte im Longrealformat durchgeführt:

var a: Lrvector[2..9]; $\{\ \mathrm{LB}(a) := N - gr = 2; \quad \mathrm{UB}(a) := N = 9;\ \}$

Im achten, gemischten Schritt wird die Multiplikation im Longrealformat ausgeführt und die Addition im 26-stelligen BCD–Format. Alle restlichen Hornerschritte laufen im 26-stelligen BCD–Format wie folgt ab:

- Die Multiplikation wird im Akkumulator ausgeführt, wobei aus Laufzeitgründen das betragsmäßig kleinste Teilprodukt weggelassen wird. Da der Akku nach dieser Multiplikation nicht ins (real|real)–Format ausgelesen wird, gilt für die relative Fehlerschranke: $\varepsilon(\text{Mul}) = 2.5002 \cdot 10^{-25}$;
- Auf das obige Multiplikationsergebnis wird im Akku der jeweilige Polynomkoeffizient rundungsfehlerfrei addiert. Diese Summe wird anschließend in das (real|real)–Format ausgelesen, weshalb die Fehlerschranke der Addition mit $\varepsilon(\text{Add}) = 5 \cdot 10^{-26}$ anzusetzen ist. Dabei muß nach Seite 46 vorausgesetzt werden, daß für jedes Ergebnis erg eines Hornerschrittes gilt: $|\text{erg}| \geq 10^{-241}$. Diese Bedingung ist in allen praktischen Fällen erfüllt, wenn $|x1 + x2|$ nicht zu klein gewählt wird und wenn das Polynom nicht in der Nähe einer seiner Nullstellen ausgewertet wird. Eine genaue Überprüfung kann mit dem **PASCAL-XSC** Programm **Horner_T** vorgenommen werden.

var b1,b2: rvector[0..1]; $\{$ LB$(b1) := 0$; UB$(b1) := N - gr - 1 = 1$; $\}$

Die oben beschriebene Polynomauswertung erfolgt durch:

procedure **HORNER** (a: Lrvector; b1,b2: rvector; x1,x2: real; var r1,r2: real;
l_add: boolean);

Der Aufruf obiger Prozedur mit den deklarierten Eingangsparametern a,b1,b2,x1,x2 liefert mit den Ausgangsgrößen r1,r2 eine Näherung $(r1 + r2) \approx \text{P}_9(x1 + x2)$ für den Polynomwert $\text{P}_9(x1 + x2)$. Es gilt:

- $r1 = 0 \implies r2 = 0; \qquad r1 <> 0 \implies |r2/(r1+r2)| < 5.0001 \cdot 10^{-13}$
- l_add = true $\iff$ Im letzten Hornerschritt wird $(b1[0] + b2[0])$ addiert.

Die Berechnung von garantierten Oberschranken des absoluten oder relativen Polynomauswertefehlers, der beim Aufruf der Prozedur **HORNER** entsteht, ist mit folgenden Programmen möglich:

IFO_BCD	IEEE–Programm, das Polynominformationen sammelt;
AWF_BCD	IEEE–Programm, das den Auswertefehler abschätzt;
Horner_T	BCD–Programm überprüft AWF_BCD–Ergebnisse.

3.5.9 Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

procedure read (**var** f: text; **var** a: rrG);
procedure write (**var** f: text; a: rrG);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung. a: rrG muß in der Form

{a.r1,a.r2,a.G} oder in der Form a.r1

eingegeben werden. Im zweiten Fall gilt: a.r2 = a.G = 0. Die Komponente a.G erhält jedoch stets den Wert $-\text{maxint} = -2147483647$, falls a.r1 verschwindet.

Erfolgt die Eingabe in der ersten Form mit beliebigen real–Werten a.r1,a.r2 , so wird auf a : rrG automatisch die Funktion Transform(a: rrG): rrG angewandt, so daß ohne Änderung des numerischen Werts von a die beiden folgenden Bedingungen von Seite 46 erfüllt werden:

$$\boxed{\text{a.r1} = 0 \quad \Longrightarrow \quad \text{a.r2} = 0 \qquad \text{(Beding. I)}}$$

$$\boxed{|\text{a.r1}| > 0 \quad \Longrightarrow \quad \left|\frac{\text{a.r2}}{\text{a.r1} + \text{a.r2}}\right| < 5.0001 \cdot 10^{-13} \qquad \text{(Beding. II)}}$$

Die Ausgabe einer rrG-Zahl erfolgt stets in der Form:

$$\{\text{a.r1,a.r2,a.G}\}$$

Beispiele:

Sei a vom Typ rrG, dann wird mit den Anweisungen read(a); write(a) und der Eingabe

$$\{\,1.234567890123\text{E}35,\ 5.934567890123\text{E}22,\ 1234567890\,\}$$

die rrG-Zahl a in der folgenden Form ausgegeben:

$$\{\,1.234567890124\text{E}{+}035,\ -4.065432109877\text{E}{+}022,\ 1234567890\,\}$$

Die Eingabe

$$\{\,4,\ -4,\ 1234567890\,\}$$

ergibt mit den gleichen obigen Anweisungen die Ausgabe des Wertes Null:

$$\{\,0.000000000000\text{E}{+}000,\ 0.000000000000\text{E}{+}000,\ -2147483647\,\}$$

Die Eingabe

$$\{\,9\text{e}{+}255,\ 9\text{e}{+}255,\ {+}2147483647\,\}$$

ergibt mit der obigen read–Anweisung eine Fehlermeldung, da der numerische Eingabewert wegen Overflow im rrG–Format nicht darstellbar ist.

Eine entsprechende Underflow–Fehlermeldung läßt sich mit den read- und write–Anweisungen jedoch nicht konstruieren!

3.5.10 Standardfunktionen

Standardfunktionen vom Typ rrG; x,y: rrG; a: Longreal			
Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
$\|x\|$	abs(x)	rrG–Bereich	0.0000
x^2	sqr(x)	$***$	$5.0001 \cdot 10^{-26}$
$x^2 - y^2$	x2_y2(x, y)	$***$	$1.5001 \cdot 10^{-25}$
$\sqrt{x}$	sqrt(x)	$x \geq 0$	$1.2253 \cdot 10^{-24}$
$\sqrt{1+x} - 1$	sqrt1pm1(x)	$x \geq -1;\ ***$	$1.8403 \cdot 10^{-24}$
$\sqrt{1-x^2}$	sqrt1mx2(x)	$\|x\| \leq 1$	$1.3004 \cdot 10^{-24}$
$\sqrt{1+x^2}$	sqrt1px2(x)	rrG–Bereich	$1.2754 \cdot 10^{-24}$
$\sqrt{x^2-1}$	sqrtx2m1(x)	$\|x\| \geq 1$	$1.3004 \cdot 10^{-24}$
$\sqrt{x^2+y^2}$	sqrtx2y2(x, y)	$x^2 + y^2 \geq 0;\ ***$	$1.2754 \cdot 10^{-24}$
e^x	exp(x)	$\|x\| < 5 \cdot 10^{+3};\ *$	$4.6213 \cdot 10^{-25}$
$e^x - 1$	expm1(x)	$x < 5 \cdot 10^{+3}$	$5.5478 \cdot 10^{-25}$
e^{-x^2}	exp_x2(x)	$\|x\| \leq 70.710516;\ **$	$4.6213 \cdot 10^{-25}$
2^x	exp2(x)	$\|x\| < 5 \cdot 10^{+3};\ *$	$4.6833 \cdot 10^{-25}$
10^x	exp10(x)	$-10^8 < x < \text{maxint};\ *$	$6.3644 \cdot 10^{-24}$
$\ln(x)$	ln(x)	$10^{-5000} \leq x < 10^{+5001}$	$2.7329 \cdot 10^{-24}$
$\ln(e \cdot x)$	ln_ex(x)	$10^{-5000} \leq x < 10^{+5001}$	$4.5584 \cdot 10^{-24}$
$\ln(1+x)$	ln1p(x)	$-1 < x < 10^{+5001}$	$2.7846 \cdot 10^{-24}$
$\ln(x^2+y^2)$	lnx2y2(x, y)	$x^2 + y^2 > 0;\ ***$	$3.2816 \cdot 10^{-24}$
$0.5 \cdot \ln(x^2+y^2)$	ln_sqrtx2y2(x, y)	$x^2 + y^2 > 0;\ ***$	$3.3317 \cdot 10^{-24}$
$\ln[(1+x)^2 + y^2]$	ln1px2y2(x, y)	$(1+x)^2+y^2>0;\ ***$	$3.8301 \cdot 10^{-24}$
$0.5 \cdot \ln[(1+x)^2+y^2]$	ln_sqrt1px2y2(x, y)	$(1+x)^2+y^2>0;\ ***$	$3.8802 \cdot 10^{-24}$
$\log_a(x)$	loga(a, x)	$10^{-5000} \leq x < 10^{+5001}$	$7.7163 \cdot 10^{-24}$
$\log_2(x)$	log2(x)	$10^{-5000} \leq x < 10^{+5001}$	$3.0830 \cdot 10^{-24}$
$\log_{10}(x)$	log10(x)	$10^{-5000} \leq x < 10^{+5001}$	$3.0830 \cdot 10^{-24}$
$\log_{10}(1+x)$	log1p(x)	$-1 < x < 10^{+5001}$	$2.8371 \cdot 10^{-24}$
x^n	power(x,n,rnd)	$n \geq 0;\ \text{rnd} = +1, -1$	

Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
x^y	power(x, y)	$\|y \cdot \ln(x)\| < 5 \cdot 10^{+3}$; $*$	$5.6503 \cdot 10^{-25}$
$\sin(x)$	$\sin(x)$	$\|x\| < 10^{+8}$	$1.0196 \cdot 10^{-24}$
$\cos(x)$	$\cos(x)$	$\|x\| < 10^{+8}$	$1.0196 \cdot 10^{-24}$
$\tan(x)$	$\tan(x)$	$\|x\| < 10^{+8}$	$2.9035 \cdot 10^{-24}$
$\cot(x)$	$\cot(x)$	$\|x\| < 10^{+8}$	$2.9035 \cdot 10^{-24}$
$\sin(\pi \cdot x)$	sinpi(x)	$\|x\| \le 2147483647$	$1.0401 \cdot 10^{-24}$
$\cos(\pi \cdot x)$	cospi(x)	$\|x\| \le 2147483647$	$1.1556 \cdot 10^{-24}$
$\tan(\pi \cdot x)$	tanpi(x)	$\|x\| < 1073741823.5$	$2.9035 \cdot 10^{-24}$
$\cot(\pi \cdot x)$	cotpi(x)	$\|x\| < 1073741823.5$	$2.9035 \cdot 10^{-24}$
$\arcsin(x)$	$\arcsin(x)$	$\|x\| \le 1$	$3.0671 \cdot 10^{-24}$
$\arccos(x)$	$\arccos(x)$	$\|x\| \le 1$	$3.1225 \cdot 10^{-24}$
$\arctan(x)$	$\arctan(x)$	rrG–Bereich	$3.0170 \cdot 10^{-24}$
$\arctan(x/y)$	arctan2(x, y)	$* * *$	$5.2675 \cdot 10^{-24}$
arccot(x)	arccot(x)	rrG–Bereich	$5.2675 \cdot 10^{-24}$
$\sinh(x)$	$\sinh(x)$	$\|x\| \le 5 \cdot 10^{+3}$	$2.6320 \cdot 10^{-24}$
$\cosh(x)$	$\cosh(x)$	$\|x\| \le 5 \cdot 10^{+3}$	$1.6875 \cdot 10^{-24}$
$\tanh(x)$	$\tanh(x)$	rrG–Bereich	$4.5980 \cdot 10^{-24}$
$\coth(x)$	$\coth(x)$	$x \neq 0$; $* * *$	$6.8486 \cdot 10^{-24}$
arsinh(x)	arsinh(x)	$\|x\| < 10^{+5001}$	$4.5070 \cdot 10^{-24}$
arcosh(x)	arcosh(x)	$1 \le x < 10^{+5001}$	$3.8908 \cdot 10^{-24}$
arcosh$(1 + x)$	arcosh1p(x)	$0 \le x < 10^{+5001}$	$6.7674 \cdot 10^{-24}$
artanh(x)	artanh(x)	$\|x\| < 1$	$4.5687 \cdot 10^{-24}$
arcoth(x)	arcoth(x)	$\|x\| > 1$	$4.6827 \cdot 10^{-24}$

$*$ Wird der angegebene Definitionsbereich nach links überschritten, so wird der Funktionswert auf Null gesetzt.

$**$ Wird der angegebene Definitionsbereich nach links und rechts überschritten, so wird der Funktionswert auf Null gesetzt.

$* * *$ Falls der rrG–Bereich wegen Overflow- oder Underflow verlassen wird, erfolgt eine entsprechende Fehlermeldung.

Standardfunktionen vom Typ rrG; x,y,a: Longreal			
Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
$\|x\|$	abs_rrG(x)	Longreal–Bereich	0.0000
x^2	sqr_rrG(x)	Longreal–Bereich	$5.0000 \cdot 10^{-26}$
$x^2 - y^2$	x2_y2_rrG(x, y)	Longreal–Bereich	$1.5001 \cdot 10^{-25}$
$\sqrt{x}$	sqrt_rrG(x)	$x \geq 0$	$1.2253 \cdot 10^{-24}$
$\sqrt{1+x} - 1$	sqrt1pm1_rrG(x)	$x \geq -1$	$1.8403 \cdot 10^{-24}$
$\sqrt{1-x^2}$	sqrt1mx2_rrG(x)	$\|x\| \leq 1$	$1.2504 \cdot 10^{-24}$
$\sqrt{1+x^2}$	sqrt1px2_rrG(x)	Longreal–Bereich	$1.2504 \cdot 10^{-24}$
$\sqrt{x^2-1}$	sqrtx2m1_rrG(x)	$\|x\| \geq 1$	$1.2504 \cdot 10^{-24}$
$\sqrt{x^2+y^2}$	sqrtx2y2_rrG(x, y)	Longreal–Bereich	$1.2504 \cdot 10^{-24}$
e^x	exp_rrG(x)	$\|x\| < 5 \cdot 10^{+3}$; $*$	$4.6213 \cdot 10^{-25}$
$e^x - 1$	expm1_rrG(x)	$x < 5 \cdot 10^{+3}$	$5.5478 \cdot 10^{-25}$
e^{-x^2}	exp_x2_rrG(x)	$\|x\| \leq 70.710516$; $**$	$4.6213 \cdot 10^{-25}$
2^x	exp2_rrG(x)	$\|x\| < 5 \cdot 10^{+3}$; $*$	$4.6833 \cdot 10^{-25}$
10^x	exp10_rrG(x)	$-10^8 < x <$ maxint; $*$	$6.3644 \cdot 10^{-24}$
$\ln(x)$	ln_rrG(x)	$x > 0$	$2.7329 \cdot 10^{-24}$
$\ln(e \cdot x)$	ln_ex_rrG(x)	$x > 0$	$4.5584 \cdot 10^{-24}$
$\ln(1+x)$	ln1p_rrG(x)	$x > -1$	$2.7846 \cdot 10^{-24}$
$\ln(x^2+y^2)$	lnx2y2_rrG(x, y)	$x^2 + y^2 > 0$	$5.1539 \cdot 10^{-24}$
$\ln(x^2+y^2)/2$	ln_sqrtx2y2_rrG(x,y)	$x^2 + y^2 > 0$	$5.2040 \cdot 10^{-24}$
$\ln[(1+x)^2+y^2]$	ln1px2y2_rrG(x, y)	$(1+x)^2 + y^2 > 0$	$2.8929 \cdot 10^{-24}$
$\ln[(1+x)^2+y^2]/2$	ln_sqrt1px2y2_rrG(x,y)	$(1+x)^2 + y^2 > 0$	$2.9430 \cdot 10^{-24}$
$\log_a(x)$	loga_rrG(a, x)	$a, x > 0;\ a \neq 1$	$7.7163 \cdot 10^{-24}$
$\log_2(x)$	log2_rrG(x)	$x > 0$	$3.0830 \cdot 10^{-24}$
$\log_{10}(x)$	log10_rrG(x)	$x > 0$	$3.0830 \cdot 10^{-24}$
$\log_{10}(1+x)$	log1p_rrG(x)	$x > -1$	$2.8371 \cdot 10^{-24}$
x^n	power_rrG(x,n,rnd)	$n \geq 0$; rnd $= +1, -1$	
x^y	power_rrG(x, y)	$\|y \cdot \ln(x)\| < 5 \cdot 10^{+3}$; $*$	$4.6212 \cdot 10^{-25}$

Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
$\sin(x)$	sin_rrG(x)	$\|x\| < 10^{+8}$	$1.0196 \cdot 10^{-24}$
$\cos(x)$	cos_rrG(x)	$\|x\| < 10^{+8}$	$1.0196 \cdot 10^{-24}$
$\tan(x)$	tan_rrG(x)	$\|x\| < 10^{+8}$	$2.9035 \cdot 10^{-24}$
$\cot(x)$	cot_rrG(x)	$\|x\| < 10^{+8}$	$2.9035 \cdot 10^{-24}$
$\sin(\pi \cdot x)$	sinpi_rrG(x)	$\|x\| \leq 2147483647$	$1.0401 \cdot 10^{-24}$
$\cos(\pi \cdot x)$	cospi_rrG(x)	$\|x\| \leq 2147483647$	$1.1556 \cdot 10^{-24}$
$\tan(\pi \cdot x)$	tanpi_rrG(x)	$\|x\| < 1073741823.5$	$2.9035 \cdot 10^{-24}$
$\cot(\pi \cdot x)$	cotpi_rrG(x)	$\|x\| < 1073741823.5$	$2.9035 \cdot 10^{-24}$
$\arcsin(x)$	arcsin_rrG(x)	$\|x\| \leq 1$	$6.5180 \cdot 10^{-24}$
$\arccos(x)$	arccos_rrG(x)	$\|x\| \leq 1$	$6.5734 \cdot 10^{-24}$
$\arctan(x)$	arctan_rrG(x)	Longreal–Bereich	$3.0170 \cdot 10^{-24}$
$\arctan(x/y)$	arctan2_rrG(x, y)	Longreal–Bereich	$5.2675 \cdot 10^{-24}$
$\text{arccot}(x)$	arccot_rrG(x)	Longreal–Bereich	$5.2675 \cdot 10^{-24}$
$\sinh(x)$	sinh_rrG(x)	$\|x\| \leq 5 \cdot 10^{+3}$	$2.6320 \cdot 10^{-24}$
$\cosh(x)$	cosh_rrG(x)	$\|x\| \leq 5 \cdot 10^{+3}$	$1.6875 \cdot 10^{-24}$
$\tanh(x)$	tanh_rrG(x)	Longreal–Bereich	$4.5980 \cdot 10^{-24}$
$\coth(x)$	coth_rrG(x)	$x \neq 0$	$6.8486 \cdot 10^{-24}$
$\text{arsinh}(x)$	arsinh_rrG(x)	Longreal–Bereich	$4.5070 \cdot 10^{-24}$
$\text{arcosh}(x)$	arcosh_rrG(x)	$x \geq 1$	$3.8513 \cdot 10^{-24}$
$\text{arcosh}(1+x)$	arcosh1p_rrG(x)	$x \geq 0$	$6.7674 \cdot 10^{-24}$
$\text{artanh}(x)$	artanh_rrG(x)	$\|x\| < 1$	$4.5687 \cdot 10^{-24}$
$\text{arcoth}(x)$	arcoth_rrG(x)	$\|x\| > 1$	$4.6827 \cdot 10^{-24}$

$*$ Wird der angegebene Definitionsbereich nach links überschritten, so wird der Funktionswert auf Null gesetzt.

$**$ Wird der angegebene Definitionsbereich nach links und rechts überschritten, so wird der Funktionswert auf Null gesetzt.

Mit Hilfe obiger Funktionen vom Ergebnistyp **rrG** und Longreal–Argumenten x können auch verschachtelte Funktionen für den Ergebnistyp Longreal **hochgenau** implementiert werden.

Als Beispiel betrachten wir $\ln(\ \cos(x)\ )$ für $0 \leq x < \pi/2$; Mit der naiven Realisierung:

```
function ln_cos(x: Longreal) : Longreal;
begin
  ln_cos := LongNext( ln( cos_rrG(x) ) );
end;
```

erhält man zum Argument $x_0 = 10^{-14}$ den im Vergleich zum exakten Funktionswert $-5.00\ldots0083\ldots\cdot 10^{-29}$ völlig unbrauchbaren Maschinenwert 0 . Dieses schlechte Ergebnis ist dadurch begründet, daß die ln-Funktion mit fehlerbehafteten cos-Funktionswerten in der Nähe ihrer Nullstelle 1 ausgewertet wird, wobei cos_rrG(x) in einer ganzen Umgebung von Null den fehlerbehafteten Funktionswert 1 liefert. Benutzt man jedoch für $x < 1$ die Identität

$$\ln(\ \cos(x)\ ) \equiv \frac{1}{2} \cdot \ln[\,1 - \sin^2(x)\,]$$

so liefert die entsprechende Implementierung

```
function ln_cos_(x: Longreal): Longreal;
begin
  if x < 1 then
  ln_cos_ := LongNext( 0.5*ln1p( -sqr(sin_rrG(x)) ) ) else
  ln_cos_ := LongNext( ln( cos_rrG(x) ) );
end;
```

für das gleiche Argument $x_0 = 10^{-14}$ den im Longreal–Format hochgenauen Funktionswert

$$\text{ln\_cos\_}(x_0) = -5.0000000000000000000 \cdot 10^{-29}$$

Fehlerabschätzung für $0 \leq x < 1$:
Mit den Bezeichnungen $\widetilde{y} := \text{sqr}(\text{sin\_rrG}(x))$, $y := \sin(x)$ gilt zunächst:

$$\begin{aligned}\widetilde{y} &= \sin^2(x) \cdot (1+\varepsilon_{sin})^2 \cdot (1+\varepsilon_{sqr}) \\ &= y \cdot (1+\varepsilon_y); \qquad |\varepsilon_y| < 2.0893 \cdot 10^{-24} = \varepsilon(y)\end{aligned}$$

$$0.5 \odot \text{ln1p}(-\widetilde{y}) = 0.5 \cdot \ln(1-\widetilde{y}) \cdot (1+\varepsilon_{ln1p}) \cdot (1+\varepsilon_M)$$
$$|\varepsilon_{ln1p}| < 2.7846 \cdot 10^{-24} = \varepsilon(\text{ln1p}); \quad |\varepsilon_M| < 5.0001 \cdot 10^{-26} = \varepsilon(M);$$

Mit der Abkürzung: $\alpha := \ln\left[1 - \dfrac{y \cdot \varepsilon_y)}{1-y}\right] \Big/ \ln(1-y)$ gilt dann die Gleichung:

$$\ln(1-\widetilde{y}) = \ln(1-y) \cdot (1+\alpha), \quad \text{und einige Abschätzungen liefern:}$$
$$|\alpha| < \varepsilon(y) \Big/ \left[1 - \frac{y}{1-y} \cdot \varepsilon(y)\right]$$
$$x < 1 \quad \rightsquigarrow \quad y = \sin^2(x) < 0.7081 \quad \rightsquigarrow \quad |\alpha| < 2.0896 \cdot 10^{-24} \quad \rightsquigarrow$$

$$\begin{aligned} 0.5 \odot \mathrm{ln1p}(-\widetilde{y}) &= 0.5 \cdot \ln(1-y) \cdot (1+\alpha) \cdot (1+\varepsilon_{ln1p}) \cdot (1+\varepsilon_M) \\ &= \ln(\cos(x)) \cdot (1+\varepsilon_1); \quad |\varepsilon_1| < 4.9243 \cdot 10^{-24} = \varepsilon(1) \end{aligned}$$

Für $x < 1$ wird also $\ln(\cos(x))$ im 26-stelligen rrG–Format mit einem relativen Höchstfehler von $4.9243 \cdot 10^{-24} = \varepsilon(1)$ berechnet. Die Funktion **LongNext** rundet diese Werte schließlich zur nächsten Longreal–Zahl, was mit einem zusätzlichen Höchstfehler von $5 \cdot 10^{-21}$ verbunden ist:

$$\begin{aligned} \mathrm{ln\_cos\_}(x) &= \ln(\cos(x)) \cdot (1+\varepsilon_1) \cdot (1+\varepsilon_{21}) \\ &= \ln(\cos(x)) \cdot (1+\varepsilon_2); \quad |\varepsilon_2| < 5.0050 \cdot 10^{-21} = \varepsilon(2) \end{aligned}$$

Damit haben wir in $x < 1$ auch eine relative Fehlerschranke für das Longreal–Format berechnet.

Fehlerabschätzung für $1 \le x < \pi/2$:
Mit den Bezeichnungen $\widetilde{y} := \mathrm{cos\_rrG}(x)$, $y := \cos(x)$ gilt zunächst:

$$\begin{aligned} \widetilde{y} = \cos(x) \cdot (1+\varepsilon_y); &\quad |\varepsilon_y| < 1.0196 \cdot 10^{-24} = \varepsilon(y) \\ \widetilde{\ln}(\widetilde{y}) = \ln(\widetilde{y}) \cdot (1+\varepsilon_{ln}); &\quad |\varepsilon_{ln}| < 2.7329 \cdot 10^{-24} \end{aligned}$$

Mit der Abkürzung: $\beta := \ln(1+\varepsilon_y)/\ln(y)$ erhält man folgende Gleichung:

$\ln(\widetilde{y}) = \ln[y(1+\varepsilon_y)] = \ln(y) \cdot (1+\beta)$, wirklich einfache Abschätzungen liefern:

$$|\beta| < \frac{\varepsilon(y)}{-\ln(\cos(1))} < 1.6563 \cdot 10^{-24} \quad \rightsquigarrow$$

$$\begin{aligned} \widetilde{\ln}(\widetilde{y}) &= \ln(y) \cdot (1+\beta) \cdot (1+\varepsilon_{ln}) \\ &= \ln(\cos(x)) \cdot (1+\varepsilon_3); \quad |\varepsilon_3| < 4.3893 \cdot 10^{-24} = \varepsilon(3) \end{aligned}$$

Für $1 \le x < \pi/2$ wird damit $\ln(\cos(x))$ im 26-stelligen rrG–Format mit einem relativen Höchstfehler von $4.3893 \cdot 10^{-24} = \varepsilon(3)$ berechnet. Die Funktion **LongNext** rundet diese Werte schließlich zur nächsten Longreal–Zahl, was mit einem zusätzlichen Höchstfehler von $5 \cdot 10^{-21}$ verbunden ist:

$$\begin{aligned} \mathrm{ln\_cos\_}(x) &= \ln(\cos(x)) \cdot (1+\varepsilon_3) \cdot (1+\varepsilon_{21}) \\ &= \ln(\cos(x)) \cdot (1+\varepsilon_4); \quad |\varepsilon_4| < 5.0044 \cdot 10^{-21} = \varepsilon(4) \end{aligned}$$

Damit haben wir in $1 \le x < \pi/2$ auch eine relative Fehlerschranke für das Longreal–Format berechnet.
Beachten Sie bitte, daß im Bereich $0 \le x < \pi/2$ die Gesamtfehlerschranke $\varepsilon(\mathrm{ln\_cos\_}) = 5.005 \cdot 10^{-21}$ ihre Gültigkeit verliert, wenn Funktionswerte in den denormalisierten Longreal–Bereich fallen, d.h. wenn die Argumente betragsmäßig zu klein gewählt werden! Der Leser möge die obige Funktion ln_cos_ so verbessern, daß das Auftreten denormalisierter Funktionswerte durch eine entsprechende Fehlermeldung verhindert wird.

3.5.11 Mathematische Konstanten

Es existieren 23 mathematische Konstanten im 26-stelligen rrG–Format , die durch Aufruf entsprechender Funktionsnamen (ohne Argument) zur Verfügung stehen:

Mathematische Konstanten im rrG – Format			
Funktionsname	**Wert**	**Fehlerschranke**	**Vergleich**
PI_rrG	π	$5.323 \cdot 10^{-27}$	PI_rrG $> \pi$
PI2_rrG	2π	$5.323 \cdot 10^{-27}$	PI2_rrG $> 2\pi$
PI_HALF_rrG	$\pi/2$	$5.323 \cdot 10^{-27}$	PI_HALF_rrG $> \pi/2$
PI_DIV4_rrG	$\pi/4$	$5.323 \cdot 10^{-27}$	PI_DIV4_rrG $> \pi/4$
R_PI_rrG	$1/\pi$	$8.011 \cdot 10^{-28}$	R_PI_rrG $> 1/\pi$
R_2PI_rrG	$1/(2\pi)$	$2.341 \cdot 10^{-27}$	R_2PI_rrG $< 1/(2\pi)$
SQRT_PI_rrG	$\sqrt{\pi}$	$9.399 \cdot 10^{-27}$	SQRT_PI_rrG $> \sqrt{\pi}$
SQRT_PI_D2_rrG	$\sqrt{\pi}/2$	$1.886 \cdot 10^{-27}$	SQRT_PI_D2_rrG $< \sqrt{\pi}/2$
R_SQRTPI_rrG	$1/\sqrt{\pi}$	$2.767 \cdot 10^{-27}$	R_SQRTPI_rrG $< 1/\sqrt{\pi}$
R_SQRTPI2_rrG	$2/\sqrt{\pi}$	$2.767 \cdot 10^{-27}$	R_SQRTPI2_rrG $< 2/\sqrt{\pi}$
R_SQRT2PI_rrG	$1/\sqrt{2\pi}$	$1.645 \cdot 10^{-28}$	R_SQRT2PI_rrG $> 1/\sqrt{2\pi}$
LN_PI_rrG	$\ln(\pi)$	$4.250 \cdot 10^{-26}$	LN_PI_rrG $> \ln(\pi)$
SQRT2_rrG	$\sqrt{2}$	$2.977 \cdot 10^{-27}$	SQRT2_rrG $> \sqrt{2}$
R_SQRT2_rrG	$1/\sqrt{2}$	$2.977 \cdot 10^{-27}$	R_SQRT2_rrG $< 1/\sqrt{2}$
LN_2_rrG	$\ln(2)$	$2.104 \cdot 10^{-27}$	LN_2_rrG $< \ln(2)$
LN_10_rrG	$\ln(10)$	$2.035 \cdot 10^{-27}$	LN_10_rrG $< \ln(10)$
R_LN_10_rrG	$1/\ln(10)$	$2.495 \cdot 10^{-27}$	R_LN_10_rrG $> 1/\ln(10)$
EUL_rrG	e	$4.977 \cdot 10^{-28}$	EUL_rrG $< e$
R_EUL_rrG	$1/e$	$4.389 \cdot 10^{-28}$	R_EUL_rrG $< 1/e$
SQRTEUL_rrG	$\sqrt{e}$	$7.392 \cdot 10^{-27}$	SQRTEUL_rrG $> \sqrt{e}$
GAMMA_C_rrG	C	$1.428 \cdot 10^{-28}$	GAMMA_C_rrG $< C$
LN_GAM_C_rrG	$\ln(C)$	$2.179 \cdot 10^{-27}$	LN_GAM_C_rrG $< C$
CATALAN_rrG	$Catalan$	$5.385 \cdot 10^{-27}$	CATALAN_rrG $< Catalan$

$$C := \lim_{m\to\infty}\left(\sum_{k=1}^{m}\frac{1}{k} - \ln(m)\right); \qquad Catalan := \sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)^2};$$

Im folgenden sind die numerischen Werte der Konstanten aus vorheriger Tabelle zusammengestellt:

x :=	**Komponenten – Werte** (x.G=0)	
PI_rrG	x.r1 = +3.141592653590	x.r2 = −2.067615373566E−13
PI2_rrG	x.r1 = +6.283185307180	x.r2 = −4.135230747132E−13
PI_H_rrG	x.r1 = +1.570796326795	x.r2 = −1.033807686783E−13
PI_DIV4_rrG	x.r1 = +0.7853981633974	x.r2 = +4.830961566085E−14
R_PI_rrG	x.r1 = +0.3183098861838	x.r2 = −9.328462232473E−15
R_2PI_rrG	x.r1 = +0.1591549430919	x.r2 = −4.664231116237E−15
SQRT_PI_rrG	x.r1 = +1.772453850906	x.r2 = −4.839727018325E−13
SQRT_PI_D2_rrG	x.r1 = +0.8862269254528	x.r2 = −4.198635091626E−14
R_SQRTPI_rrG	x.r1 = +0.5641895835478	x.r2 = −4.371305192055E−14
R_SQRTPI2_rrG	x.r1 = +1.128379167096	x.r2 = −4.874261038411E−13
R_SQRT2PI_rrG	x.r1 = +0.3989422804014	x.r2 = +3.267793994606E−14
LN_PI_rrG	x.r1 = +1.144729885849	x.r2 = +4.001741434274E−13
SQRT2_rrG	x.r1 = +1.414213562373	x.r2 = +9.504880168872E−14
R_SQRT2_rrG	x.r1 = +0.7071067811865	x.r2 = +4.752440084436E−14
LN_2_rrG	x.r1 = +0.6931471805599	x.r2 = +4.530941723212E−14
LN_10_rrG	x.r1 = +2.302585092994	x.r2 = +4.568401799145E−14
R_LN_10_rrG	x.r1 = +0.4342944819033	x.r2 = −4.817234887108E−14
EUL_rrG	x.r1 = +2.718281828459	x.r2 = +4.523536028747E−14
R_EUL_rrG	x.r1 = +0.3678794411714	x.r2 = +4.232159552377E−14
SQRTEUL_rrG	x.r1 = +1.648721270700	x.r2 = +1.281468486508E−13
GAMMA_C_rrG	x.r1 = +0.5772156649015	x.r2 = +3.286060651209E−14
LN_GAM_C_rrG	x.r1 = −0.5495393129816	x.r2 = −4.482233766177E−14
CATALAN_rrG	x.r1 = +0.9159655941772	x.r2 = +1.901505460351E−14

3.6 Der Datentyp lrG

Bei der Implementierung **hochgenauer** Standardfunktionen im Longreal–Format kann es z.B. bei der power-Funktion notwendig werden, Zwischenrechnungen mit mehr als 26 Dezimalstellen durchzuführen. Analog zum rrG–Format bietet sich dann an, eine Longreal- und real–Zahl mit insgesamt 34 Dezimalstellen aneinanderzuhängen, was durch den Datentyp **lrG** realisiert wird

```
Type lrG = record
           l : Longreal;  r : real;
           G : integer
           end;
```

der durch das Standardmodul STDMOD zur Verfügung gestellt wird.
Durch a : lrG und

$$a.l := 1.23456789012345678901; \quad a.r := 2.345678901234E{-}21; \quad a.G := +1234567;$$

wird eine BCD-Zahl mit mindestens 34 Mantissenstellen dargestellt, deren Exponentenbereich nur durch die integer–Zahlen begrenzt ist:

$$a = 1.2345678901234567890123456789 01234 \cdot 10^{+1234567}$$
$$-2147483648 \le a.G \le +2147483647$$

Für die korrekte Durchführung der arithmetischen Grundoperationen muß ein lrG–Operand a die beiden folgenden Bedingungen erfüllen:

$$\boxed{a.l = 0 \quad \Longrightarrow \quad a.r = 0 \qquad \text{(Beding. I)}}$$

$$\boxed{|a.l| > 0 \quad \Longrightarrow \quad \left|\frac{a.r}{a.l + a.r}\right| < 5.0001 \cdot 10^{-21} \qquad \text{(Beding. II)}}$$

Die beiden obigen Bedingungen werden dabei automatisch realisiert, falls wie in allen praktischen Anwendungen a.l und a.r durch die folgenden Anweisungen aus dem Akkumulator ausgelesen werden:

```
a.l  := LongNext( Akku );  { a   : lrG;  Akku: dotprecision }
Akku := Akku - a.l;        { a.l : Longreal;                }
a.r  := #*( Akku );        { a.r : real;                    }
```

Bedeutet Akku den numerischen Wert des Akkumulators, so gilt mit den obigen Anweisungen zusätzlich:

$$\boxed{|a.l| \ge 10^{-233}, \quad (a.l + a.r) = \text{Akku} \cdot (1 + \varepsilon) \qquad \Longrightarrow \qquad |\varepsilon| \le 5 \cdot 10^{-34}}$$

Das zweimalige Auslesen des Akkumulators erfolgt also mit dem relativen Höchstfehler $\varepsilon(34) := 5{\cdot}10^{-34}$, falls $|\text{a.l}| \geq 10^{-233}$. Nach den drei obigen **PASCAL-XSC** Anweisungen erreicht man daher mit der Abfrage

```
if (SIGN(Akku) <> 0) and (EXPO(a.l) < TEST_34)
                then  Fehlermeldung
```

daß das zweimalige Auslesen des Akkumulators in die lrG–Komponenten a.l und a.r den relativen Fehler $5 \cdot 10^{-34}$ nicht übersteigt; TEST_34 = −232 wird vom Modul STDMOD als real-Konstante exportiert.

Der Anwender findet für den Datentyp **lrG** nur einen Minimalsatz von Hilfsfunktionen und Operatoren, die zur Realisierung der benötigten Standardfunktionen in 34-stelliger Dezimalarithmetik wirklich erforderlich waren. Da Intervallrechnungen im lrG–Format nicht vorgesehen sind, fehlen z.B. bei den Grundoperatoren alle gerichteten Rundungen.

3.6.1 Monadische Operatoren

Für den Datentyp **lrG** stehen die monadischen Operatoren **+ −** in gewohnter Weise zur Verfügung.

3.6.2 Addition, Subtraktion

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Variablen vom Typ lrG addieren bzw. subtrahieren. Falls beide Operanden den gleichen Zehnerexponenten besitzen, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen.

- procedure **ADD_21** (xl: Longreal; x,y: real; var rl: Longreal; var r: real);
  $(\text{rl} + \text{r}) = (\text{xl} + \text{x} + \text{y}) \cdot (1 + \varepsilon), \quad |\text{rl}| \geq 10^{-233} \implies |\varepsilon| \leq 5 \cdot 10^{-34}$,
  falls kein Overflow auftritt. $|\text{r}/(\text{rl} + \text{r})| < 5.0001 \cdot 10^{-21}$ wird erfüllt.

- procedure **ADD_22** (xl: Longreal; x: real; yl: Longreal; y: real;
  var rl: Longreal; var r: real);
  $(\text{rl} + \text{r}) = (\text{xl} + \text{x} + \text{yl} + \text{y}) \cdot (1 + \varepsilon), \quad |\text{rl}| \geq 10^{-233} \implies |\varepsilon| \leq 5 \cdot 10^{-34}$,
  falls kein Overflow auftritt. $|\text{r}/(\text{rl} + \text{r})| < 5.0001 \cdot 10^{-21}$ wird erfüllt.

- operator **+** (x,y: lrG) s: lrG;

  Vorauss.: $|\ \text{x.r}/(\text{x.l} + \text{x.r})\ | < 5.0001 \cdot 10^{-21}$
  $|\ \text{y.r}/(\text{y.l} + \text{y.r})\ | < 5.0001 \cdot 10^{-21}$
  $\text{s} = (\text{x} + \text{y}) \cdot (1 + \varepsilon)$
  $\implies \quad |\varepsilon| \leq 5.0001 \cdot 10^{-34}; \quad |\text{s.r}/(\text{s.l} + \text{s.r})| < 5.0001 \cdot 10^{-21}$

- operator **−** (x,y: lrG) d: lrG;

  Vorauss.: $|\ \text{x.r}/(\text{x.l} + \text{x.r})\ | < 5.0001 \cdot 10^{-21}$
  $|\ \text{y.r}/(\text{y.l} + \text{y.r})\ | < 5.0001 \cdot 10^{-21}$
  $\text{d} = (\text{x} - \text{y}) \cdot (1 + \varepsilon)$
  $\implies \quad |\varepsilon| \leq 5.0001 \cdot 10^{-34}; \quad |\text{d.r}/(\text{d.l} + \text{d.r})| < 5.0001 \cdot 10^{-21}$

3.6.3 Multiplikation

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Variablen vom Typ rrG multiplizieren. Falls die Summe der Zehnerexponenten beider Operanden verschwindet, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen. Es existieren auch die wichtigsten Operatoren mit gemischten Operanden.

- procedure **MUL_12** (x: real; yl: Longreal; y: real;
  var rl: Longreal; var r: real);
  $(\mathrm{rl}+\mathrm{r}) = \mathrm{x}\cdot(\mathrm{yl}+\mathrm{y})\cdot(1+\varepsilon)$, $|\mathrm{rl}| \geq 10^{-233}$
  $\Longrightarrow$ $|\varepsilon| \leq 5\cdot 10^{-34}$, $|\mathrm{r}/(\mathrm{rl}+\mathrm{r})| < 5.0001\cdot 10^{-21}$ wird erfüllt.

- procedure **MUL_12** (xl: Longreal; yl: Longreal; y: real;
  var rl: Longreal; var r: real);
  $(\mathrm{rl}+\mathrm{r}) = \mathrm{xl}\cdot(\mathrm{yl}+\mathrm{y})\cdot(1+\varepsilon)$, $|\mathrm{rl}| \geq 10^{-233}$
  $\Longrightarrow$ $|\varepsilon| \leq 5\cdot 10^{-34}$, $|\mathrm{r}/(\mathrm{rl}+\mathrm{r})| < 5.0001\cdot 10^{-21}$ wird erfüllt.

- procedure **MUL_22** (xl: Longreal; x: real; yl: Longreal; y: real;
  var rl: Longreal; var r: real);
  Vorauss.: $|\,\mathrm{x}/(\mathrm{xl}+\mathrm{x})\,| < 5.0001\cdot 10^{-21}$
  $|\,\mathrm{y}/(\mathrm{yl}+\mathrm{y})\,| < 5.0001\cdot 10^{-21}$
  $(\mathrm{rl}+\mathrm{r}) = (\mathrm{xl}+\mathrm{x})\cdot(\mathrm{yl}+\mathrm{y})\cdot(1+\varepsilon)$, $|\mathrm{rl}| \geq 10^{-233}$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-34}$; $|\mathrm{r}/(\mathrm{rl}+\mathrm{r})| < 5.0001\cdot 10^{-21}$

- operator ***** (x,y: lrG) p: lrG;
  Vorauss.: $|\,\mathrm{x.r}/(\mathrm{x.l}+\mathrm{x.r})\,| < 5.0001\cdot 10^{-21}$
  $|\,\mathrm{y.r}/(\mathrm{y.l}+\mathrm{y.r})\,| < 5.0001\cdot 10^{-21}$
  $\mathrm{p} = (\mathrm{x}\cdot\mathrm{y})\cdot(1+\varepsilon)$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-34}$; $|\mathrm{p.r}/(\mathrm{p.l}+\mathrm{p.r})| < 5.0001\cdot 10^{-21}$

3.6.4 Division

Mit Hilfe der folgenden Prozeduren und Operatoren lassen sich Quotienten vom Typ lrG berechnen. Falls die Zehnerexponenten beider lrG–Operanden übereinstimmen, sind aus Laufzeitgründen die Prozeduren den Operatoren vorzuziehen.

- procedure **DIV_22** (xl: Longreal; x: real; yl: Longreal; y: real;
  var rl: Longreal; var r: real);
  Vorauss.: $|\,\mathrm{x}/(\mathrm{xl}+\mathrm{x})\,| < 5.0001\cdot 10^{-21}$
  $|\,\mathrm{y}/(\mathrm{yl}+\mathrm{y})\,| < 5.0001\cdot 10^{-21}$
  $(\mathrm{rl}+\mathrm{r}) = (\mathrm{xl}+\mathrm{x})/(\mathrm{yl}+\mathrm{y})\cdot(1+\varepsilon)$, $|\mathrm{rl}| \geq 10^{-233}$
  $\Longrightarrow$ $|\varepsilon| \leq 5.0001\cdot 10^{-34}$; $|\mathrm{r}/(\mathrm{rl}+\mathrm{r})| < 5.0001\cdot 10^{-21}$

- operator **/** (x,y: lrG) q: lrG;

$$
\begin{array}{lll}
\text{Vorauss.:} & |\ \mathrm{x.r}/(\mathrm{x.l} + \mathrm{x.r})\ | \quad < \quad 5.0001 \cdot 10^{-21} & \\
 & |\ \mathrm{y.r}/(\mathrm{y.l} + \mathrm{y.r})\ | \quad < \quad 5.0001 \cdot 10^{-21} & \\
 & \mathrm{q} = (\mathrm{x}/\mathrm{y}) \cdot (1 + \varepsilon) & \\
\Longrightarrow & |\varepsilon| \leq 5.0002 \cdot 10^{-34}; & |\mathrm{q.r}/(\mathrm{q.l} + \mathrm{q.r})| < 5.0001 \cdot 10^{-21}
\end{array}
$$

3.6.5 Vergleiche

Mit Hilfe der beiden folgenden Funktionen sind lediglich zwei Vergleiche vorgesehen:

- function **greater_21** (xl: Longreal; x,a: real): boolean;
  Vorauss.: $|\ \mathrm{x}/(\mathrm{xl} + \mathrm{x})\ | \quad < \quad 5.0001 \cdot 10^{-21}$
  $\mathrm{greater\_21} = \mathrm{true} \quad \Longleftrightarrow \quad (\mathrm{xl} + \mathrm{x}) > \mathrm{a}$

- function **less_21** (xl: Longreal; x,a: real): boolean;
  Vorauss.: $|\ \mathrm{x}/(\mathrm{xl} + \mathrm{x})\ | \quad < \quad 5.0001 \cdot 10^{-21}$
  $\mathrm{less\_21} = \mathrm{true} \quad \Longleftrightarrow \quad (\mathrm{xl} + \mathrm{x}) < \mathrm{a}$

3.6.6 Transferfunktionen und Zuweisungsoperatoren

Die folgenden Prozeduren und Operatoren ermöglichen die Verbindung zwischen den Datentypen **rrG, lrG** :

- procedure **Trans** (rl: Longreal; r: real; var x1,x2: real);
  Vorauss.: $|\mathrm{r}/(\mathrm{rl} + \mathrm{r})| < 5.0001 \cdot 10^{-21}; \quad (\mathrm{x1} + \mathrm{x2}) = (\mathrm{rl} + \mathrm{r}) \cdot (1 + \varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 5 \cdot 10^{-26}; \qquad |\mathrm{x2}/(\mathrm{x1} + \mathrm{x2})| < 5.0001 \cdot 10^{-13}$

- procedure **Trans** (x1,x2: real; var rl: Longreal; var r: real);
  Vorauss.: $|\mathrm{x2}/(\mathrm{x1} + \mathrm{x2})| < 5.0001 \cdot 10^{-13}; \quad (\mathrm{rl} + \mathrm{r}) = (\mathrm{x1} + \mathrm{x2}) \cdot (1 + \varepsilon)$
  $\Longrightarrow \quad |\varepsilon| = 0; \qquad |\mathrm{r}/(\mathrm{rl} + \mathrm{r})| < 5.0001 \cdot 10^{-21}$

- operator **:=** (var a: rrG; b: lrG);
  Vorauss.: $|\mathrm{b.r}/(\mathrm{b.l} + \mathrm{b.r})| < 5.0001 \cdot 10^{-21}; \qquad \mathrm{a} = \mathrm{b} \cdot (1 + \varepsilon)$
  $\Longrightarrow \quad |\varepsilon| \leq 5 \cdot 10^{-26}; \qquad |\mathrm{a.r2}/(\mathrm{a.r1} + \mathrm{a.r2})| < 5.0001 \cdot 10^{-13}$

- operator **:=** (var a: lrG; b: rrG);
  Vorauss.: $|\mathrm{b.r2}/(\mathrm{b.r1} + \mathrm{b.r2})| < 5.0001 \cdot 10^{-13}; \qquad \mathrm{a} = \mathrm{b} \cdot (1 + \varepsilon)$
  $\Longrightarrow \quad |\varepsilon| = 0; \qquad |\mathrm{a.r}/(\mathrm{a.l} + \mathrm{a.r})| < 5.0001 \cdot 10^{-21}$

3.6.7 Hilfsfunktionen

Bei den auftretenden Funktionsparametern vom Typ **lrG** wird vorausgesetzt, daß die Bedingungen I,II von Seite 66 erfüllt sind. Dies wird immer dann der Fall sein, wenn die Komponenten dieser Parameter durch das auf der gleichen Seite beschriebene zweimalige Auslesen des Akkumulators ins Longreal- bzw. real–Format definiert sind. Besitzen Funktionen den Ergebnistyp lrG, so erfüllen auch die Komponenten aller Funktionswerte automatisch die Bedingungen I,II von Seite 66.

- function **Expo** (x: lrG) : integer;
  Berechnet den Zehnerexponenten analog zu Expo(x: real): integer;

- procedure **Mant_34** (xl: Longreal; x: real; var rl: Longreal: var r: real);
  Berechnet die Mantisse von (xl + x) analog zu Mant(x: real): real , falls $|x/(xl + x)| < 5.0001 \cdot 10^{-21}$ erfüllt ist.
  Es gilt also: $|rl + r| = 0$ oder $0.1 \leq |rl + r| < 1$;

- function **Scale** (x: lrG): lrG;
  y := Scale(x) liefert folgende Ergebnisse:

  1. y = x, '=' im Sinne der Numerik;
  2. y.l = 0 $\Longrightarrow$ y.r = 0, y.G = −Maxint = −2147483647;
  3. $0.1 \leq |y.l| < 1$, falls y.l <> 0; (Skalierung);
  4. $|y.r/(y.l + y.r)| < 5.0001 \cdot 10^{-21}$, falls y.l <> 0;

  Falls das Ergebnis y = x in extremen Ausnahmefällen nicht erfüllt werden kann, erfolgt eine entsprechende Fehlermeldung.

- function **Transform** (x: lrG): lrG;
  Vorauss.: x.l, x.r, x.G seien beliebig;
  Durch y := Transform(x) wird x so transformiert, daß gilt:

  1. y = x;
  2. y.l = 0 $\Longrightarrow$ y.r = 0;
  3. y.l <> 0 $\Longrightarrow$ $|y.r/(y.l + y.r)| < 5.0001 \cdot 10^{-21}$;

3.6.8 Hornerschema

Gegeben sei das Polynom vom Grade $N = 9$:

$$\mathrm{P}_9(x) = (bl[0] + b[0]) + (bl[1] + b[1]) \cdot x^1 + a[2] \cdot x^2 + \ldots + a[9] \cdot x^9$$

das für das 34-stellige BCD–Argument $x = (xl + x)$ nach dem Hornerschema auszuwerten ist. Die Zahlen xl, x müssen folgende Bedingungen erfüllen:

$$xl = 0 \quad \Longrightarrow \quad x = 0; \qquad xl <> 0 \quad \Longrightarrow \quad |x/(xl + x)| < 5.0001 \cdot 10^{-21}$$

Aus Laufzeitgründen werden die ersten $gr = 7$ groben Hornerschritte im Longrealformat durchgeführt:

var a: Lrvector[2..9]; { LB(a) := $N - gr = 2$; UB(a) := $N = 9$; }

Im achten, gemischten Schritt wird die Multiplikation im Longrealformat ausgeführt und die Addition im 34-stelligen BCD–Format. Alle restlichen Hornerschritte laufen im 34-stelligen BCD–Format wie folgt ab:

- Die Multiplikation wird im Akkumulator ausgeführt, wobei aus Laufzeitgründen das betragsmäßig kleinste Teilprodukt weggelassen wird. Da der Akku nach dieser Multiplikation nicht ins (Longreal|real)–Format ausgelesen wird, gilt für die relative Fehlerschranke: $\varepsilon(\text{Mul}) = 2.5002 \cdot 10^{-41}$;

- Auf das obige Multiplikationsergebnis wird im Akku der jeweilige Polynomkoeffizient rundungsfehlerfrei addiert. Diese Summe wird anschließend in das (Longreal|real)–Format ausgelesen, weshalb die Fehlerschranke der Addition mit $\varepsilon(\text{Add}) = 5 \cdot 10^{-34}$ anzusetzen ist. Dabei muß nach Seite 66 vorausgesetzt werden, daß für jedes Ergebnis erg eines Hornerschrittes gilt: $|\text{erg}| \geq 10^{-233}$.
  Diese Bedingung ist in allen praktischen Fällen erfüllt, wenn $|xl + x|$ nicht zu klein gewählt wird und wenn das Polynom nicht in der Nähe einer seiner Nullstellen ausgewertet wird. Eine genaue Überprüfung kann mit dem **PASCAL-XSC** Programm **HORN_TS** vorgenommen werden.

var bl: Lrvector[0..1]; b: rvector[0..1];
{ LB(bl) := 0; UB(bl) := $N - gr - 1 = 1$; }

Die oben beschriebene Polynomauswertung erfolgt durch:

procedure **HORNER** (a,bl: Lrvector; b: rvector; xl: Longreal; x: real;
var rl: Longreal; var r: real);

Der Aufruf obiger Prozedur mit den deklarierten Eingangsparametern a,bl,b,xl,x liefert mit den Ausgangsgrößen rl,r eine Näherung $(rl + r) \approx \text{P}_9(xl + x)$ für den Polynomwert $\text{P}_9(xl + x)$. Es gilt:

- $rl = 0 \implies r = 0$; $r1 <> 0 \implies |r/(rl + r)| < 5.0001 \cdot 10^{-21}$

Die Berechnung von garantierten Oberschranken des absoluten oder relativen Polynomauswertefehlers, der beim Aufruf der Prozedur **HORNER** entsteht, ist mit folgenden Programmen möglich:

IFO_BCDS	IEEE–Programm, das Polynominformationen sammelt;
AWF_BCDS	IEEE–Programm, das den Auswertefehler abschätzt;
HORN_TS	BCD–Programm überprüft AWF_BCDS–Ergebnisse.

3.6.9 Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

procedure read (**var** f: text; **var** a: lrG);
procedure write (**var** f: text; a: lrG);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung. a: lrG muß in der Form

{a.l,a.r,a.G} oder in der Form a.l

eingegeben werden. Im zweiten Fall gilt: a.r = a.G = 0. Die Komponente a.G erhält jedoch stets den Wert $-\text{maxint} = -2147483647$, falls a.l verschwindet.

Erfolgt die Eingabe in der ersten Form mit beliebigen Werten a.l,a.r , so wird auf a : lrG automatisch die Funktion Transform(a: lrG): lrG angewandt, so daß ohne Änderung des numerischen Werts von a die beiden folgenden Bedingungen von Seite 66 erfüllt werden:

$$\boxed{\text{a.l} = 0 \quad \Longrightarrow \quad \text{a.r} = 0 \qquad \text{(Beding. I)}}$$

$$\boxed{|\text{a.l}| > 0 \quad \Longrightarrow \quad \left|\frac{\text{a.r}}{\text{a.l} + \text{a.r}}\right| < 5.0001 \cdot 10^{-21} \qquad \text{(Beding. II)}}$$

Die Ausgabe einer lrG-Zahl erfolgt stets in der Form:

$$\{\text{a.l,a.r,a.G}\}$$

Beispiele:

Sei a vom Typ lrG, dann wird mit den Anweisungen read(a); write(a) und der Eingabe

$$\{\, 1.23456789012345678901\text{E}35,\ 5.934567890123\text{E}14,\ -1234567890 \,\}$$

die lrG-Zahl a in der folgenden Form ausgegeben:

$$\{\, 1.23456789012345678902\text{E}{+}035,\ -4.065432109877\text{E}{+}014,\ -1234567890 \,\}$$

Die Eingabe

$$\{\, 4\text{e}{+}500,\ -4e + 500,\ 1234567890 \,\}$$

ergibt mit den gleichen obigen Anweisungen die Ausgabe des Wertes Null:

$$\{\, 0.000000000000\text{E}{+}000,\ 0.000000000000\text{E}{+}000,\ -2147483647 \,\}$$

Die Eingabe

$$\{\, 9\text{e}{+}511,\ 9\text{e}{+}511,\ +2147483647 \,\}$$

ergibt mit der obigen read–Anweisung eine Fehlermeldung, da der numerische Eingabewert wegen Overflow im lrG–Format nicht darstellbar ist.

Eine entsprechende Underflow–Fehlermeldung läßt sich mit den read- und write–Anweisungen jedoch nicht konstruieren!

3.6.10 Standardfunktionen

Standardfunktionen vom Typ lrG; x: lrG			
Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
$\sqrt{x}$	sqrt(x)	$x \geq 0$	$6.6850 \cdot 10^{-31}$
$\sqrt{1-x^2}$	sqrt1mx2(x)	$\lvert x\rvert \leq 1$	$6.6926 \cdot 10^{-31}$
$\sqrt{1+x^2}$	sqrt1px2(x)	rrG–Bereich	$6.6901 \cdot 10^{-31}$
$\sqrt{x^2-1}$	sqrtx2m1(x)	$\lvert x\rvert \geq 1$	$6.6926 \cdot 10^{-31}$
$\ln(x)$	ln(x)	$10^{-5000} \leq x < 10^{+5001}$	$2.3299 \cdot 10^{-30}$
$\log_{10}(x)$	log10(x)	$10^{-5000} \leq x < 10^{+5001}$	$2.3310 \cdot 10^{-30}$

Standardfunktionen vom Typ lrG; x: Longreal			
Funktion	**Aufruf**	**Argumentbereich**	**Rel. Fehler**
$\sqrt{1-x^2}$	sqrt1mx2_lrG(x)	$\lvert x\rvert \leq 1$	$6.6876 \cdot 10^{-31}$
$\sqrt{1+x^2}$	sqrt1px2_lrG(x)	Longreal–Bereich	$6.6927 \cdot 10^{-31}$
$\sqrt{x^2-1}$	sqrtx2m1_lrG(x)	$\lvert x\rvert \geq 1$	$6.6876 \cdot 10^{-31}$

Die Komponenten x.l,x.r eines Funktionsarguments x: lrG müssen die beiden Bedingungen von Seite 66 erfüllen:

$$\boxed{\mathrm{x.l} = 0 \quad \Longrightarrow \quad \mathrm{x.r} = 0 \qquad \text{(Beding. I)}}$$

$$\boxed{|\mathrm{x.l}| > 0 \quad \Longrightarrow \quad \left|\frac{\mathrm{x.r}}{\mathrm{x.l} + \mathrm{x.r}}\right| < 5.0001 \cdot 10^{-21} \qquad \text{(Beding. II)}}$$

Ist y: lrG das Ergebnis einer der obigen Funktionen, so gelten die beiden obigen Bedingungen **automatisch** auch für die Komponenten y.l,y.r .

Kapitel 4

Intervallrechnung

Intervallarithmetiken existieren bzgl. der Grunddatentypen real, Longreal, rrG und dotprecision.

4.1 Intervallrechnung mit dem Typ real

Mit dem Modul **I_ARI** werden die für das Rechnen mit reellen Intervallen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

4.1.1 Der Datentyp interval

Der Datentyp *interval* ist definiert durch:

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

Mit der Deklaration: **var** x : interval; bedeutet x dann das reelle Intervall:

$x = [x.\mathrm{INF}, x.\mathrm{SUP}]$

4.1.2 Operatoren

Sämtliche in diesem Modul vordefinierten arithmetischen und Verbands–Operatoren liefern den Ergebnistyp *interval*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der Rundung zum kleinsten einschließenden Intervall vom Typ *interval* zur Verfügung.

Ist bei den Grundoperatoren $+, -, *, /$ ein Operand vom Typ *interval*, so sind für den zweiten Operanden die folgenden Typen möglich:

interval, integer oder **real**.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind mengentheoretisch zu interpretieren. Dabei bedeutet:

$=$ gleich	$<>$ ungleich	$<$ echte Teilmenge von
$<=$ Teilmenge von	$>$ echte Obermenge von	$>=$ Obermenge von

Der Operator **in** steht für die Relation „liegt in“ zwischen einem integer/real– und einem *interval*–Operanden oder für die Relation „echt enthalten in“ zwischen zwei *interval*–Operanden zur Verfügung.
Es gilt:

x	**in**	y	$\Longleftrightarrow$	(x.inf > y.inf)	**and**	(x.sup <> y.sup).
3.2	**in**	y	$\Longleftrightarrow$	(3.2 $\geq$ y.inf)	**and**	(3.2 $\leq$ y.sup).

Der Operator >< testet auf Disjunktheit zweier Intervalle. Dabei heißen zwei Intervalle x, y disjunkt, wenn gilt $x \cap y = \emptyset$ (leere Menge).
Die Verbandsoperatoren +* bzw. ** bezeichnen die Bildung der Intervall–Hülle bzw. des Durchschnitts, d.h. der Operator +* liefert das kleinste, beide Operanden umfassende Intervall, und der Operator ** liefert das Schnittintervall. Ein leerer Schnitt führt zu einem Laufzeitfehler.

rechter Operand	integer real	interval	Bemerkungen
linker Operand			
monadisch		+, −	
integer real	+*	$\diamond$ **in**, =, <> +*	$\diamond \in \{+, -, *, /\}$
interval	$\diamond$ =, <> +*	$\diamond$ **in**, $\vee$, >< +*, **	$\vee \in \{=, <>, <, <=, >, >=\}$

Tabelle 4.1: Die Operatoren des Moduls I_ARI

Beispiele:

Seien a, b vom Typ *interval* mit: $a = [-1, 3]$; $b = [3, 4]$, dann liefern die Operatoren +, −, *, /, ><, +*, ** die folgenden Ergebnisse:

Ausdruck	**Ergebnis**	**Ausdruck**	**Ergebnis**
a + b	$[2, 7]$	a +* b	$[-1, 4]$
a − b	$[-5, 0]$	a ** b	$[3, 3]$
a * b	$[-4, 12]$	a >< b	*false*
a / b	$[-0.3333333333334, 1]$		

4.1.3 Transferfunktionen

Zur Wandlung zwischen den Typen *real* und *interval* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
intval (r1,r2)	*interval*	Intervall mit $inf = r1$ und $sup = r2$; $r1 \leq r2$
intval (r)	*interval*	Punktintervall mit $inf = sup = r$
inf (x)	*real*	Untergrenze von x
sup (x)	*real*	Obergrenze von x

r, r1, r2 = *real*–Ausdruck; x = *interval*–Ausdruck

4.1.4 Überladungen des Zuweisungsoperators

Die Wandlung von *real* nach *interval* wird auch in Form einer überladenen Zuweisung bereitgestellt:

Zuweisung	Bedeutung
x := r	x := intval (r)

x = *interval*–**Variable**; r = *real*–Ausdruck

4.1.5 Standardfunktionen

Alle für *interval*–Argumente verfügbaren, hochgenauen Standardfunktionen sind in Tabelle 4.2 zusammengestellt.

Darüberhinaus sind Funktionen für die Berechnung des Mittelpunktes, des Vorzeichens und des Durchmessers von Intervallen verfügbar:

- Intervall–Vorzeichen: $\operatorname{sign}(x) = \left\{ \begin{array}{rl} -1 & \text{x.sup} < 0 \\ 0 & 0 \in x \\ +1 & \text{x.inf} > 0 \end{array} \right\}$
- Intervall–Durchmesser: $\operatorname{diam}(x) := \sup(x) \mathrel{->} \inf(x);$
- Die Mittelpunktsfunktion ergibt sich aus der Anweisung:

  $\operatorname{mid}(x) := \# * (0.5 * \inf(x) + 0.5 * \sup(x))\,;$

Funktion	**Aufruf**	**Funktion**	**Aufruf**
$\lvert x\rvert$	abs(x)	cos(x)	cos(x)
x^2	sqr(x)	tan(x)	tan(x)
$x^2 - y^2$	x2_y2(x, y)	cot(x)	cot(x)
$\sqrt{x}$	sqrt(x)	$\sin(\pi \cdot x)$	sinpi(x)
$\sqrt{1+x} - 1$	sqrt1pm1(x)	$\cos(\pi \cdot x)$	cospi(x)
$\sqrt{x^2 + y^2}$	sqrtx2y2(x, y)	$\tan(\pi \cdot x)$	tanpi(x)
$\sqrt{1 + x^2}$	sqrt1px2(x)	$\cot(\pi \cdot x)$	cotpi(x)
$\sqrt{1 - x^2}$	sqrt1mx2(x)	arcsin(x)	arcsin(x)
$\sqrt{x^2 - 1}$	sqrtx2m1(x)	arccos(x)	arccos(x)
e^x	exp(x)	arctan(x)	arctan(x)
$e^x - 1$	expm1(x)	arctan2(x, y)	arctan2(x, y)
e^{-x^2}	exp_x2(x)	arccot(x)	arccot(x)
2^x	exp2(x)	sinh(x)	sinh(x)
10^x	exp10(x)	cosh(x)	cosh(x)
x^y	power(x, y)	tanh(x)	tanh(x)
$\ln(x)$	ln(x)	coth(x)	coth(x)
$\ln(e \cdot x)$	ln_ex(x)	arsinh(x)	arsinh(x)
$\ln(1 + x)$	ln1p(x)	arcosh(x)	arcosh(x)
$\ln(x^2 + y^2)/2$	ln_sqrtx2y2(x, y)	arcosh($1 + x$)	arcosh1p(x)
$\ln[(1 + x)^2 + y^2)]/2$	ln_sqrt1px2y2(x, y)	artanh(x)	artanh(x)
$\log_a(x)$	loga(a, x)	arcoth(x)	arcoth(x)
$\log_2(x)$	log2(x)	sign(x)	sign(x)
$\log_{10}(x)$	log10(x)	diam(x)	diam(x)
$\log_{10}(1 + x)$	log1p(x)	mid(x)	mid(x)
$\sin(x)$	sin(x)	blow(x, eps)	blow(x, eps)

Tabelle 4.2: Mathematische Standardfunktionen des Moduls I_ARI;
x, y: *interval*–Ausdrücke; eps,a: *real*–Ausdrücke;

- Mit Ausnahme von sign(x), diam(x), und mid(x) liefern alle anderen Funktionen aus Tabelle 4.2 ein Ergebnis vom Typ *interval* mit **hochgenau** berechneten Intervallgrenzen, d.h. nur in ganz seltenen Ausnahmefällen wird z.B. die untere Intervallgrenze um 1 ulp zu klein berechnet.

- Bei Einschließungsmethoden wird die Funktion *blow* für die sogenannte Epsilonaufblähung eines Intervalls benötigt. Mit einem *real*–Ausdruck eps und einem *interval*–Ausdruck x wird die blow(x, eps)–Funktion aufgebaut durch die Anweisungen:

$$\begin{array}{lll} \text{y} & := & (1+\text{eps})*\text{x}-\text{eps}*\text{x}; \\ \text{blow (x,eps)} & := & \text{intval (pred (inf(y)), succ (sup(y)));} \end{array}$$

- $y := \text{abs}(x);$ $\{|r| \,\big|\, r \in x \cap \mathbb{R}\} = y$
- $y := \text{sqr}(x);$ $\{r^2 \,\big|\, r \in x \cap \mathbb{R}\} \subseteq y$
- $y := \text{sqrt}(x);$ $\{\sqrt{r} \,\big|\, r \in x \cap \mathbb{R},\ \inf(x) \geq 0\} \subseteq y$

Beispiele:

Seien a, b, c vom Typ *interval*, erzeugt durch die Anweisungen:

$a := \text{intval}(-1, 3);$ $b := \text{intval}(1, 4)$ c:=intval (-5,-4);

dann liefern die Funktionen:

abs, sqr, mid, sign, sqrt, diam, x2_y2, sqrtx2y2, arctan2

die folgenden Ergebnisse:

Ausdruck	**Ergebnis**	**Ausdruck**	**Ergebnis**
abs (a)	$[0, 3]$	sqr (b)	$[1; 16]$
abs (b)	$[1, 4]$	mid (a)	1
sqr (a)	$[0, 9]$	diam (a)	4
sign (a)	0	sign (b)	1
sign (c)	-1	sqrt (b)	$[1, 2]$
x2_y2 (a,b)	$[-16, 8]$	sqrtx2y2 (a,b)	$[1, 5]$
arctan2 (b,a)	$[-1.57..95; 1.57..95]$	arctan2 (a,b)	$[-0.78..75; 1.24..99]$

4.1.6 Ein-/Ausgabeprozeduren

Es stehen die bekannten Prozeduren

procedure read (**var** f: text; **var** a: interval);
procedure write (**var** f: text; a: interval);

mit optionalem Fileparameter, beliebig vielen Ein–/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Intervalls $x = [a, b]$ muß in der Form

$[a, b]$ oder in der Form a

erfolgen. Der zweite Fall dient zur vereinfachten Eingabe eines Punktintervalls $x = [a, a]$.

Die Ausgabe eines Intervalls durch die write–Prozedur erfolgt stets in der Form

$$[a, b]$$

Das Ausgabeformat ist dabei durch folgende Regelung bestimmt:

> Ist x eine Variable vom Typ *interval* und stimmen die Werte von inf (x) und sup (x) auf den ersten $n \geq 2$ Ziffern überein, so gibt die Anweisung write (x) die Unterschranke a und die Oberschranke b jeweils auf n gerundete Mantissenstellen aus.
> Für $n = 0$ und $n = 1$ werden a und b stets auf zwei Mantissenstellen gerundet.

Werden also durch write (x) $n > 2$ Mantissenziffern ausgegeben, so weiß der Anwender, daß inf (x) und sup (x) genau in den ersten n Ziffern übereinstimmen!

In folgender Tabelle sind für den Wert einer Variablen x vom Typ *interval* die jeweiligen Ergebnisse der Ausgabeprozedur write (x) zusammengestellt:

$x = [\inf(x), \sup(x)]$	write(x)
$[2, 3]$	[2.0E+000, 3.0E+000]
$[1.1, 1.2]$	[1.1E+000, 1.2E+000]
$[1.11, 1.13]$	[1.1E+000, 1.2E+000]
$[2.000, 2.0001]$	[2.000E+000, 2.001E+000]

Die *real*–Werte inf (x) und sup (x) können natürlich auch (**ohne** zusätzliche Rundungen durch die write (x)–Prozedur) wie folgt ausgegeben werden:

- write (inf(x), sup(x));
- write (x.inf, x.sup);

x := intval $(1.11, 1.13)$ und write (x.inf,x.sup) ergeben:

[1.110000000000E+000, 1.13000000000E+000]

im Gegensatz zur entsprechenden Ausgabe von write (x) in obiger Tabelle.

4.2 Intervallrechnung mit dem Typ Longreal

Mit dem Modul **LI_ARI** werden die für das Rechnen mit reellen Intervallen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

4.2.1 Der Datentyp Linterval

Der Datentyp *Linterval* ist definiert durch:

type Linterval = **record** inf, sup : Longreal **end**;

Mit der Deklaration: **var** x : Linterval; bedeutet x dann das reelle Intervall:

x = [x.INF, x.SUP]

4.2.2 Operatoren

Wir betrachten zunächst alle im Modul LI_ARI vordefinierten arithmetischen und Verbands–Operatoren mit dem Ergebnistyp *Linterval*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der Rundung zum jeweils kleinsten einschließenden Intervall vom Typ *Linterval* zur Verfügung.

Ist bei den Grundoperatoren $+, -, *, /$ ein Operand vom Typ *Linterval*, so sind für den zweiten Operanden die folgenden Typen möglich:

integer, real, Longreal, interval oder **Linterval**.

Die Vergleichsoperatoren = <> < <= > >= sind mengentheoretisch zu interpretieren. Dabei bedeutet:

= gleich	<> ungleich	< echte Teilmenge von
<= Teilmenge von	> echte Obermenge von	>= Obermenge von

Der Operator **in** steht wieder für die Relation „liegt in" zwischen einem integer/real/Longreal– und einem *Linterval*–Operanden oder für die Relation „echt enthalten in" zwischen zwei *Linterval*–Operanden zur Verfügung. Es gilt:

$$\begin{array}{lllllll} \text{x} & \textbf{in} & \text{y} & \Longleftrightarrow & (\text{x.inf} > \text{y.inf}) & \textbf{and} & (\text{x.sup} <> \text{y.sup}). \\ 3.2 & \textbf{in} & \text{y} & \Longleftrightarrow & (3.2 \geq \text{y.inf}) & \textbf{and} & (3.2 \leq \text{y.sup}). \end{array}$$

Der Operator >< testet auf Disjunktheit zweier Intervalle. Dabei heißen zwei Intervalle x, y disjunkt, wenn gilt $x \cap y = \emptyset$ (leere Menge).
Die Verbandsoperatoren $+*$ bzw. $**$ bezeichnen die Bildung der Intervall–Hülle bzw. des Durchschnitts, d.h. der Operator $+*$ liefert das kleinste, beide Operanden umfassende Intervall, und der Operator $**$ liefert das Schnittintervall. Ein leerer Schnitt führt zu einem Laufzeitfehler.

rechter Operand	integer real Longreal	Linterval interval	Bemerkungen
linker Operand	Ergebnistypen: *Linterval* oder *boolean*		
monadisch		$+, -$	
integer real Longreal		$\diamond$ **in**, $=$, $<>$ $+*$	$\diamond \in \{+, -, *, /\}$
Linterval interval	$\diamond$ $=$, $<>$ $+*$	$\diamond$ **in**, $\vee$, $><$ $+*$, $**$	$\vee \in \{=, <>, <, <=, >, >=\}$

Beispiele:

Seien a, b vom Typ *Linterval* mit: $a = [-1, 3]$; $b = [3, 4]$, dann liefern die Operatoren $+, -, *, /, ><, +*, **$ die folgenden Ergebnisse:

Ausdruck	**Ergebnis**	**Ausdruck**	**Ergebnis**
a + b	$[2, 7]$	a +* b	$[-1, 4]$
a − b	$[-5, 0]$	a ** b	$[3, 3]$
a * b	$[-4, 12]$	a >< b	*false*
a / b	$[-0.3333333333333333333334, 1]$		

4.2.3 Transferfunktionen

In nachfolgender Tabelle sind alle die Funktionen angegeben, die den Zusammenhang zwischen dem Typ Linterval einerseits und den Typen integer, real, Longreal und interval andererseits herstellen:

Funktion	**Ergebnistyp**	**Bedeutung**
SHORT (Lx)	interval	Lx $\subseteq$ SHORT (Lx)
LONG (x)	*Linterval*	Long(x) $\equiv$ x, aber Datentypen verschieden!
LINTVAL (a,b)	*Linterval*	Intervall mit $inf = a$ und $sup = b$; $a \leq b$
LINTVAL (a)	*Linterval*	Punktintervall mit $inf = sup = a$
INF (Lx)	*Longreal*	Untergrenze von Lx
SUP (Lx)	*Longreal*	Obergrenze von Lx

a, b = *integer/real/Longreal*–Ausdruck
Lx = *Linterval*–Ausdruck x = *interval*–Ausdruck;

4.2.4 Standardfunktionen

Dem Anwender stehen die in nachfolgender Tabelle 4.3 zusammengestellten hochgenauen, mathematischen Intervall–Standardfunktionen zur Verfügung. Argumente x,y vom Typ Linterval liefern Funktionswerte ebenfalls vom Typ Linterval.

Funktion	**Aufruf**	**Funktion**	**Aufruf**
$\|x\|$	abs(x)	cos(x)	cos(x)
x^2	sqr(x)	tan(x)	tan(x)
$x^2 - y^2$	x2_y2(x, y)	cot(x)	cot(x)
$\sqrt{x}$	sqrt(x)	$\sin(\pi \cdot x)$	sinpi(x)
$\sqrt{1+x} - 1$	sqrt1pm1(x)	$\cos(\pi \cdot x)$	cospi(x)
$\sqrt{x^2 + y^2}$	sqrtx2y2(x, y)	$\tan(\pi \cdot x)$	tanpi(x)
$\sqrt{1 + x^2}$	sqrt1px2(x)	$\cot(\pi \cdot x)$	cotpi(x)
$\sqrt{1 - x^2}$	sqrt1mx2(x)	arcsin(x)	arcsin(x)
$\sqrt{x^2 - 1}$	sqrtx2m1(x)	arccos(x)	arccos(x)
e^x	exp(x)	arctan(x)	arctan(x)
$e^x - 1$	expm1(x)	arctan2(x, y)	arctan2(x, y)
e^{-x^2}	exp_x2(x)	arccot(x)	arccot(x)
2^x	exp2(x)	sinh(x)	sinh(x)
10^x	exp10(x)	cosh(x)	cosh(x)
x^y	power(x, y)	tanh(x)	tanh(x)
$\ln(x)$	ln(x)	coth(x)	coth(x)
$\ln(e \cdot x)$	ln_ex(x)	arsinh(x)	arsinh(x)
$\ln(1 + x)$	ln1p(x)	arcosh(x)	arcosh(x)
$\ln(x^2 + y^2)/2$	ln_sqrtx2y2(x, y)	arcosh($1 + x$)	arcosh1p(x)
$\ln[(1 + x)^2 + y^2)]/2$	ln_sqrt1px2y2(x, y)	artanh(x)	artanh(x)
$\log_a(x)$	loga(a, x)	arcoth(x)	arcoth(x)
$\log_2(x)$	log2(x)	sign(x)	sign(x)
$\log_{10}(x)$	log10(x)	diam(x)	diam(x)
$\log_{10}(1 + x)$	log1p(x)	mid(x)	mid(x)
$\sin(x)$	sin(x)	blow(x, eps)	blow(x, eps)

Tabelle 4.3: Hochgenaue Standardfunktionen des Moduls LI_ARI;
x, y: Linterval; eps: real; a: Longreal;

Die Funktionen mid, diam sind für die Berechnung des Mittelpunktes und des Durchmessers von Intervallen vorgesehen.

Die Mittelpunktsfunktion liefert mit: m := MID (x) den Longreal–Mittelpunkt m eines Intervalls x. Kann m mit dem Akkumulator nicht maximalgenau berechnet werden, erfolgt eine entsprechende Fehlermeldung mit Programmabbruch.

Der Intervalldurchmesser ist definiert durch: $\mathrm{diam}(x) := \sup(x) -> \inf(x)$;

Bei Einschließungsmethoden wird oft die Funktion **blow** für die sogenannte Epsilonaufblähung benötigt. Mit einem *real*–Ausdruck eps und einem *Linterval*–Ausdruck x wird die *blow*–Funktion aufgebaut durch die Anweisungen:

y	:=	(Long(1) + eps) ∗ x − eps ∗ x;
blow (x,eps)	:=	intval (pred (inf(y)), succ (sup(y)));

Beispiele:

Seien a, b vom Typ *Linterval*, erzeugt durch die Anweisungen:

$$a := \text{intval}\ (-1, 3); \quad b := \text{intval}\ (2);$$

dann liefern die Funktionen abs, sqr, mid, diam die folgenden Ergebnisse:

Ausdruck	**Ergebnis**	**Ausdruck**	**Ergebnis**
abs (a)	$[0, 3]$	sqr (b)	$[4; 4]$
abs (b)	$[2, 2]$	mid (a)	1
sqr (a)	$[0, 9]$	diam (a)	4

Anmerkung:

Mit Hilfe der obigen Intervallfunktionen vom Typ *Linterval* können **hochgenaue** Intervallfunktionen vom Typ *interval* sehr einfach bereitgestellt werden, sofern dies nicht schon nach Tabelle 4.2 geschehen ist. Das folgende Beispiel zeigt, wie z.B. die Potenzfunktion x^y für interval–Argumente x, y zu deklarieren ist:

```
program test (input,output);
use i_ari, li_ari;
function power (x,y: interval): interval;
begin   power := SHORT ( power(Long(x),Long(y)) );   end;
begin { test }  ...   end.
```

4.2.5 Überladungen des Zuweisungsoperators

Die Wandlung von *integer/real/Longreal/interval* nach *Linterval* wird auch in Form einer überladenen Zuweisung bereitgestellt:

Zuweisung	Bedeutung
Lx := r	Lx := Lintval (r)
Lx := x	Lx := intval (r1,r2)

Lx = *Linterval*–**Variable**; x = *interval*–**Ausdruck**;
r = *integer/real/Longreal*–**Ausdruck**; r1,r2 = *integer/real*–**Ausdruck**;

4.2.6 Ein-/Ausgabeprozeduren

Es stehen die bekannten Prozeduren

> **procedure** read (**var** f: text; **var** a: Linterval);
> **procedure** write (**var** f: text; a: Linterval);

mit optionalem Fileparameter, beliebig vielen Ein–/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Intervalls $x = [a, b]$ muß in der Form

$[a, b]$ oder in der Form a

erfolgen. Der zweite Fall dient zur vereinfachten Eingabe eines Punktintervalls $x = [a, a]$.

Die Ausgabe eines Intervalls durch die write–Prozedur erfolgt stets in der Form

$$[a, b]$$

Das Ausgabeformat ist dabei durch folgende Regelung bestimmt:

> Ist Lx eine Variable vom Typ *Linterval* und stimmen die Werte von inf (Lx) und sup (Lx) auf den ersten $n \geq 2$ Ziffern überein, so gibt die Anweisung write (Lx) die Unterschranke a und die Oberschranke b jeweils auf n gerundete Mantissenstellen aus.
> Für $n = 0$ und $n = 1$ werden a und b stets auf zwei Mantissenstellen gerundet.

Werden also durch write (Lx) $n > 2$ Mantissenziffern ausgegeben, so weiß der Anwender, daß inf (Lx) und sup (Lx) genau in den ersten n Ziffern übereinstimmen!

In folgender Tabelle sind für den Wert einer Variablen Lx vom Typ *Linterval* die jeweiligen Ergebnisse der Ausgabeprozedur write (Lx) zusammengestellt:

$Lx = [\inf(Lx), \sup(Lx)]$	write(Lx)
$[2, 3]$	[2.0E+000, 3.0E+000]
$[1.1, 1.2]$	[1.1E+000, 1.2E+000]
$[1.11, 1.13]$	[1.1E+000, 1.2E+000]
$[2.000, 2.0001]$	[2.000E+000, 2.001E+000]

Die *Longreal*–Werte inf (Lx) und sup (Lx) können natürlich auch (**ohne** zusätzliche Rundungen durch die write (Lx)–Prozedur) wie folgt ausgegeben werden:

- write (inf(Lx), sup(Lx));
- write (Lx.inf, Lx.sup);

Lx := Lintval $(1.11, 1.13)$ und write (Lx.inf,Lx.sup) ergeben:

[1.11000000000000000000E+000, 1.13000000000000000000E+000]

im Gegensatz zur entsprechenden Ausgabe von write (Lx) in obiger Tabelle.

4.2.7 Exakte Auswertung von Intervallausdrücken

In **PASCAL–XSC** können mit Hilfe der Operatoren $+, -, *$ Intervall–Ausdrücke vom Typ **interval** mit Hilfe des langen Akkus rundungsfehlerfrei berechnet und durch die Anweisung

i := ##(exakter *Intervallausdruck*);

mit nur einer einzigen Rundung pro Intervallgrenze in eine Variable i vom Typ interval abgespeichert werden. Dieses ##–Konzept steht auch für die BCD–Version in **vollem Umfang** zur Verfügung; es ist jedoch **nicht** anwendbar, wenn nur einer der Operanden im exakt auszuwertenden *Intervallausdruck* vom Typ Longreal oder Linterval ist!

Um das ##–Konzept für diese Datentypen wenigstens simulieren zu können, benötigt man den Datentyp **idotprecision**:

type idotprecision = **record** INF, SUP: dotprecision **end**;

4.2.7.1 Operatoren

Wie auf Seite 42 bechrieben, muß auch jetzt z.B. für einen Additionsoperator mit Linterval–Operanden und einem Ergebnis vom Typ idotprecision ein **neuer** Operatorname (plus_i) gewählt werden, da der normale + Operator mit den gleichen Operanden schon für den Ergebnistyp Linterval vergeben ist. Beachten Sie, daß Operatoren, Prozeduren und Funktionen vom Compiler nicht bzgl. des Ergebnistyps unterschieden werden! In nachfolgender Tabelle 4.4 sind die entsprechenden neuen Operatornamen zusammengestellt:

Operation	Operator–Name	Ergebnistyp
Addition	**plus_i**	idotprecision
Subtraktion	**minus_i**	idotprecision
Multiplikation	**times_i**	idotprecision

Tabelle 4.4: Operatornamen mit Ergebnistyp idotprecision

In Tabelle 4.5 sind für die Operatoren **plus_i, minus_i, times_i,** $*$ die erlaubten Operanden–Typen angegeben. Mit einem *dotprecision*–Operanden existiert jedoch der $*$ Operator nur mit dem zweiten Operandentyp idotprecision; vergl. auch Tabelle 4.6 !

rechter Operand	integer/real Longreal	interval Linterval	idotprecision (dotprecision)
linker Operand			
integer/real Longreal	times_i plus_i, minus_i	times_i plus_i, minus_i	$*$ plus_i, minus_i
interval Linterval	times_i plus_i, minus_i	times_i plus_i, minus_i	$*$ plus_i, minus_i
idotprecision (dotprecision)	$*$ plus_i, minus_i	$*$ plus_i, minus_i	$*$ plus_i, minus_i

Tabelle 4.5: Operatoren des Moduls LI_ARI mit Ergebnistyp idotprecision

Wertzuweisungen an eine Variable vom Typ idotprecision:
Durch Überladen des Zuweisungsoperators := sind folgende Wertzuweisungen möglich:

idotprecision–Variable := Linterval–Ausdruck;
idotprecision–Variable := interval–Ausdruck;
idotprecision–Variable := dotprecision–Ausdruck;
idotprecision–Variable := Longreal–Ausdruck;
idotprecision–Variable := real/integer–Ausdruck;

In nachfolgender Tabelle 4.6 können zu den Operatoren mit den Ergebnistypen:

boolean, interval, Linterval, idotprecision

alle möglichen Operanden abgelesen werden:

rechter Operand	integer real Longreal	Linterval interval	idotprecision	dotprecision
linker Operand				
monadisch		$+, -$		
integer real Longreal		$\diamond, \triangle$ **in**, =, <> +*	$\square$ **in**, =, <> +*	
Linterval interval	$\diamond, \triangle$ =, <> +*	$\diamond, \triangle$ **in**, $\vee$, >< +*, **	$\square$ **in**, $\vee$, >< +*, **	
idotprecision	$\square$ =, <> +*	$\square$ **in**, $\vee$, >< +*, **	$\square$ **in**, $\vee$, >< +*, **	$\square$
dotprecision			$\square$	

Tabelle 4.6: Operatoren vom Typ: *interval,Linterval,idotprecision* oder *boolean*

Bedeutung der Operatoren:

1. $+, -$; Dies sind monadische Operatoren mit den Ergebnistypen: *interval, Linterval, idotprecision*;
2. $\diamond \in \{+, -, *, /\}$, Grundoperatoren mit Ergebnistyp: *interval, Linterval*;
3. $\square \in$ {plus_i, minus_i, $*$}; Additions-, Subtraktions- und Multiplikationsoperatoren mit Ergebnistyp *idotprecision*;
4. $\triangle \in$ {plus_i, minus_i, times_i}; Additions-, Subtraktions- und Multiplikationsoperatoren mit Ergebnistyp *idotprecision*;
5. **in** -Operator testet: $[a, b] \subset [A, B]$ oder $x \in [A, B]$;
6. >< Operator testet auf Disjunktheit zweier Intervalle;
7. +* Operator liefert die konvexe Hülle zweier Intervalle oder eines Intervalls mit einer Zahl;
8. ** Operator liefert das Schnittintervall; ein leeres Schnittintervall erzeugt einen Laufzeitfehler;
9. $\vee \in \{=, <>, <, <=, >, >=\}$, Vergleichsoperatoren zwischen Intervallen;

4.2.7.2 Transferfunktionen

Zur Wandlung zwischen den Typen

integer/real/Longreal/dotprecision/interval/Linterval

einerseits und dem Typ *idotprecision* andererseits werden folgende Transferfunktionen bereitgestellt: idxsup (Funktion)

Funktion	Ergebnistyp	Bedeutung
DOTINTVAL(d1,d2)	*idotprecision*	Intervall mit $inf = d1$ und $sup = d2$;
DOTINTVAL(d)	*idotprecision*	Punktintervall mit $inf = sup = d$;
DOTINTVAL(r1,r2)	*idotprecision*	Intervall mit $inf = r1$ und $sup = r2$;
DOTINTVAL(r)	*idotprecision*	Punktintervall mit $inf = sup = r$;
INF (di)	*dotprecision*	Untergrenze von di
SUP (di)	*dotprecision*	Obergrenze von di
LONG (di)	*Linterval*	di–Einschlg. durch Intervall: Linterval
SHORT (di)	*interval*	di–Einschlg. durch Intervall: interval

Tabelle 4.7: Transferfunktionen mit dem Typ idotprecision

r,r1,r2 = *integer/real/Longreal*–Ausdruck, r1 $\leq$ r2, sonst Fehlermeldung;
di = *idotprecision*–Ausdruck;
d,d1,d2 = *dotprecision*–Variable, d1 $\leq$ d2, sonst Fehlermeldung;

4.2.7.3 Simulation von SUM–Ausdrücken

Zur Simulation von **SUM**–Ausdrücken im ##–Konzept stehen die folgenden Datentypen und Prozeduren zur Verfügung:

- **type** rvector = **dynamic** array [*] of **real**;
- **type** Lrvector = **dynamic** array [*] of **Longreal**;
- **type** ivector = **dynamic** array [*] of **interval**;
- **type** Livector = **dynamic** array [*] of **Linterval**;

1. procedure **ADDNACCU** (var di: idotprecision; var a: rvector);
   Exakte Addition des real-Feldes a zum Intervall di;
2. procedure **ADDNACCU** (var di: idotprecision; var a: Lrvector);
   Exakte Addition des Longreal-Feldes a zum Intervall di;
3. procedure **ADDNACCU** (var di: idotprecision; var a: ivector);
   Exakte Addition des interval-Feldes a zum Intervall di;

4. procedure **ADDNACCU** (var di: idotprecision; var a: Livector);
   Exakte Addition des Linterval-Feldes a zum Intervall di;
5. procedure **SUBNACCU** (var di: idotprecision; var a: rvector);
   Exakte Subtraktion des real-Feldes a vom Intervall di;
6. procedure **SUBNACCU** (var di: idotprecision; var a: Lrvector);
   Exakte Subtraktion des Longreal-Feldes a vom Intervall di;
7. procedure **SUBNACCU** (var di: idotprecision; var a: ivector);
   Exakte Subtraktion des interval-Feldes a vom Intervall di;
8. procedure **SUBNACCU** (var di: idotprecision; var a: Livector);
   Exakte Subtraktion des Linterval-Feldes a vom Intervall di;
9. procedure **PADDNACCU** (var di: idotprecision; var a,b: rvector);
   Exakte Addition des real-Skalarproduktes zum Intervall di;
10. procedure **PADDNACCU** (var di: idotprecision; var a,b: Lrvector);
   Exakte Addition des Longreal-Skalarproduktes zum Intervall di;
11. procedure **PADDNACCU** (var di: idotprecision; var a,b: ivector);
   Exakte Addition des interval-Skalarproduktes zum Intervall di;
12. procedure **PADDNACCU** (var di: idotprecision; var a,b: Livector);
   Exakte Addition des Linterval-Skalarproduktes zum Intervall di;
13. procedure **PSUBNACCU** (var di: idotprecision; var a,b: rvector);
   Exakte Subtraktion des real-Skalarproduktes vom Intervall di;
14. procedure **PSUBNACCU** (var di: idotprecision; var a,b: Lrvector);
   Exakte Subtraktion des Longreal-Skalarproduktes vom Intervall di;
15. procedure **PSUBNACCU** (var di: idotprecision; var a,b: ivector);
   Exakte Subtraktion des interval-Skalarproduktes vom Intervall di;
16. procedure **PSUBNACCU** (var di: idotprecision; var a,b: Livector);
   Exakte Subtraktion des Linterval-Skalarproduktes vom Intervall di;

4.2.7.4 Beispiele

1. Beispiel:

Mit den Deklarationen:
```
var doti : idotprecision;
    a,b  : Livector [1..10];
    Li   : Linterval;
```

sind z.B. folgende Anweisungen möglich:

```
doti := 0;   PADDNACCU (doti,a,b);
Li := LONG( doti  plus_i  a[1] times_i b[10]  minus_i  a[2] times_i b[9] );
```

Das Intervall Li liefert eine Einschließung der exakten Summe mit **nur einer Rundung** bzgl. Intervallunter– und –Obergrenze!

2. Beispiel:

Mit Hilfe der Operatoren $*$ und **plus_i** zwischen *idotprecision*- und *Linterval*-Operanden kann ein Polynom

$$\mathrm{P}(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

mit folgender Funktion **rundungsfehlerfrei** berechnet werden:

```
function HORNER_IDOT(var a:Livector; x:Linterval):idotprecision;
var k : integer;    y : idotprecision;
begin
     y := a[ub(a)];
     for k := ub(a)−1 downto lb(a) do
     y := y ∗ x plus_i a[k];    { Intervall–Hornerschema }
     HORNER_IDOT := y
end;
```

Die Koeffizienten

$a_0 = -0.01;$	$a_1 = +0.39;$	$a_2 = -7.02;$	$a_3 = +77.22;$
$a_4 = -579.15;$	$a_5 = +3127.41;$	$a_6 = -12509.64;$	$a_7 = +37528.92;$
$a_8 = -84440.07;$	$a_9 = +140733.45;$	$a_{10} = -168880.14;$	$a_{11} = +138174.66;$
$a_{12} = -69087.33;$	$a_{13} = +15943.23;$		

realisieren das Polynom $\mathrm{P}(x) = \frac{1}{100} \cdot (3x-1)^{13}$ mit der Nullstelle $x_0 = \frac{1}{3}$;
Mit den Deklarationen:

```
var a: Livector[0..13];    x,y: Linterval;    doti: idotprecision;
```

und den Anweisungen:

```
x   := 0.3333333333333333;    {x ≈ x0;  x ist Punktintervall!}
y   := LONG( HORNER_IDOT (a,x) );
```

erhält man mit der BCD–Version das **exakte Intervallergebnis:**

$$y = \mathrm{P}(x) \in [-1.00\ldots00 \cdot 10^{-210}; -1.00\ldots00 \cdot 10^{-210}]$$

Führt man ganz entsprechende Rechnungen mit der Binärversion von PASCAL–XSC durch, wobei die a_i und das Argument x vorher durch Binärintervalle einzuschließen sind, so erhält man:

$$\mathrm{P}(x1) \in [-4.41283\ldots593 \cdot 10^{-15}; +2.40837\ldots365 \cdot 10^{-15}]$$

Die enorme Abweichung von **195 Größenordnungen** von der exakten BCD–Einschließung ist alleine bedingt durch den Umstand, daß **vor** der eigentlichen rundungsfehlerfreien Polynomauswertung neben dem Argument x auch alle Polynomkoeffizienten in Binärintervalle eingeschlossen werden müssen, deren Durchmesser i.a. von Null verschieden sind, da Dezimalzahlen i.a. nicht exakt als Binärzahlen endlicher Länge darstellbar sind.

4.2.8 Wertebereich monotoner Funktionen

Zur Realisierung der hochgenauen Intervallfunktionen aus Tabelle 4.3 mit Hilfe monotoner Punktfunktionen vom Ergebnistyp **rrG** sind im Modul **LI_ARI** folgende Werkzeuge bereitgestellt:

- function **Lintval_f** ({ Wertebereichs–Einschließg. }
  function f(x: Longreal): rrG; { Monotone Punktfunktion }
  I: Linterval; { Monotonie–Intervall }
  up: boolean; { Monotoniebeschreibung }
  eps: real { Vergrößerte Fehlerschranke }
) : Linterval; { Typ des Wertebereichs }

  Die Funktion f(x: Longreal): rrG muß im Intervall I monoton sein. Mit der relativen Fehlerschranke $\varepsilon(f)$ der Funktion f muß gelten:

  $$\text{eps} := 1.00001 *\!> \varepsilon(f);$$

  Ist $f(x)$ in I monoton wachsend (fallend), so ist der Parameter up true (false) zu wählen. Die Anweisung

  ```
  y := Lintval_f(f,I,up,eps);
  ```

  liefert dann mit y: Linterval eine hochgenaue Einschließung aller Funktionswerte $f(x)$, mit $x \in$ I.

- function **Lintval_frrG** ({ Wertebereichs–Einschließg. }
  function f(x: rrG): rrG; { Monotone Punktfunktion }
  I: Linterval; { Monotonie–Intervall }
  up: boolean; { Monotoniebeschreibung }
  eps: real { Vergrößerte Fehlerschranke }
) : Linterval; { Typ des Wertebereichs }

  Die Funktion f(x: rrG): rrG muß im Intervall I monoton sein. Mit der relativen Fehlerschranke $\varepsilon(f)$ der Funktion f muß gelten:

  $$\text{eps} := 1.00001 *\!> \varepsilon(f);$$

  Ist $f(x)$ in I monoton wachsend (fallend), so ist der Parameter up true (false) zu wählen. Die Anweisung

  ```
  y := Lintval_frrG(f,I,up,eps);
  ```

  liefert dann mit y: Linterval eine hochgenaue Einschließung aller Funktionswerte $f(x)$, mit $x \in$ I.

4.3 Intervallrechnung mit dem Typ rrG

Mit dem Modul **RRGI_ARI** werden nur die Operatoren, Funktionen und Prozeduren bereitgestellt, die zur Implementierung von mathematischen Intervallfunktionen notwendig sind. Es ist nicht vorgesehen, die komplette Intervallrechnung wie etwa im Modul **LI_ARI** zu realisieren, so daß z.B. Vergleichsoperatoren für Intervalle oder Intervall–Grundoperationen mit gemischten Operandentypen nur in dem Umfang bereitgestellt werden, wie sie tatsächlich erforderlich waren.

4.3.1 Der Datentyp rrGinterval

Der Datentyp *rrGinterval* ist definiert durch:

type rrGinterval = **record** inf, sup : rrG **end**;

Mit der Deklaration: **var** x : rrGinterval; bedeutet x dann das reelle Intervall:

$$\mathrm{x} = [\mathrm{x.INF}, \mathrm{x.SUP}]$$

4.3.2 Operatoren

Für Operanden vom Typ rrGinterval stehen die monadischen Operatoren $+\ -$ und die arithmetischen Operatoren $+\ -\ *\ /$ zur Verfügung. Bei der Multiplikation kann zusätzlich der erste Operand vom Typ rrG,Longreal,real und bei der Division der zweite Operand ebenfalls vom Typ rrG,Longreal,real gewählt werden.
Als Vergleichsoperatoren sind lediglich die Operatoren $= <>$ definiert, wobei beide Operanden vom Typ rrGinterval sein müssen.
Beim Operator **in** ist der zweite Operand vom Typ rrGinterval, und der erste Operand kann vom Typ real,rrGinterval gewählt werden.
Der Operator $><$ testet zwei rrGinterval–Operanden auf Disjunktheit, wobei zwei Intervalle disjunkt heißen, wenn sie kein gemeinsames Element enthalten.
Die Verbandsoperatoren $+*\ \ **$ sind als Intervall-Hülle bzw. als Durchschnitt zweier rrGintervall–Operanden definiert.

4.3.3 Transferfunktionen

In nachfolgender Tabelle sind alle die Funktionen angegeben, die den Zusammenhang zwischen dem Typ rrGinterval einerseits und den Typen integer, real, Longreal und rrG andererseits herstellen:

Funktion	**Ergebnistyp**	**Bedeutung**
RRGINTVAL (a,b)	rrGinterval	$[a, b]$; $a, b:$ rrG
RRGINTVAL (a)	rrGinterval	$[a, a]$; $a:$ integer/real/Longreal/rrG

4.3.4 Standardfunktionen

Es stehen die in nachfolgender Tabelle 4.8 angegebenen mathematischen Intervall–Standardfunktionen zur Verfügung. Argumente x,y vom Typ rrGinterval liefern, mit Ausnahme von sign(x), Funktionswerte ebenfalls vom Typ rrGinterval.

$$\mathrm{sign}(x) = \begin{cases} 0 & \text{falls } 0 \in x \\ +1 & \text{falls } x.\mathrm{inf} > 0 \\ -1 & \text{falls } x.\mathrm{sup} < 0 \end{cases}$$

Funktion	**Aufruf**	**Funktion**	**Aufruf**
$\|x\|$	abs(x)	sin(x)	sin(x)
x^2	sqr(x)	cos(x)	cos(x)
$x^2 - y^2$	x2_y2(x, y)	tan(x)	tan(x)
$\sqrt{x}$	sqrt(x)	cot(x)	cot(x)
$\sqrt{1+x} - 1$	sqrt1pm1(x)	$\sin(\pi \cdot x)$	sinpi(x)
$\sqrt{x^2+y^2}$	sqrtx2y2(x, y)	$\cos(\pi \cdot x)$	cospi(x)
$\sqrt{1+x^2}$	sqrt1px2(x)	$\tan(\pi \cdot x)$	tanpi(x)
$\sqrt{1-x^2}$	sqrt1mx2(x)	$\cot(\pi \cdot x)$	cotpi(x)
$\sqrt{x^2-1}$	sqrtx2m1(x)	arcsin(x)	arcsin(x)
e^x	exp(x)	arccos(x)	arccos(x)
$e^x - 1$	expm1(x)	arctan(x)	arctan(x)
e^{-x^2}	exp_x2(x)	arctan2(x, y)	arctan2(x, y)
2^x	exp2(x)	arccot(x)	arccot(x)
10^x	exp10(x)	sinh(x)	sinh(x)
x^y	power(x, y)	cosh(x)	cosh(x)
$\ln(x)$	ln(x)	tanh(x)	tanh(x)
$\ln(e \cdot x)$	ln_ex(x)	coth(x)	coth(x)
$\ln(1+x)$	ln1p(x)	arsinh(x)	arsinh(x)
$\ln(x^2+y^2)/2$	ln_sqrtx2y2(x, y)	arcosh(x)	arcosh(x)
$\ln[(1+x)^2+y^2)]/2$	ln_sqrt1px2y2(x, y)	arcosh($1+x$)	arcosh1p(x)
$\log_a(x)$	loga(a, x)	artanh(x)	artanh(x)
$\log_2(x)$	log2(x)	arcoth(x)	arcoth(x)
$\log_{10}(x)$	log10(x)	sign(x)	sign(x)
$\log_{10}(1+x)$	log1p(x)	diam(x)	diam(x)

Tabelle 4.8: Mathematische Standardfunktionen des Moduls RRGI_ARI;
x, y: rrGinterval; a: Longreal;

Um bei verschachtelten Intervall–Funktionen Zwischenrechnungen auch im genaueren rrGinterval–Format durchführen zu können, sind in nachfolgender Tabelle 4.9 alle mathematischen Intervall–Standardfunktionen angegeben, die für Argumente x,y vom Typ Linterval Funktionswerte vom Typ rrGinterval liefern:

Funktion	**Aufruf**	**Funktion**	**Aufruf**
$\lvert x\rvert$	abs_rrG(x)	sin(x)	sin_rrG(x)
x^2	sqr_rrG(x)	cos(x)	cos_rrG(x)
$x^2 - y^2$	x2_y2_rrG(x, y)	tan(x)	tan_rrG(x)
$\sqrt{x}$	sqrt_rrG(x)	cot_rrG(x)	cot_rrG(x)
$\sqrt{1+x} - 1$	sqrt1pm1_rrG(x)	sin($\pi \cdot x$)	sinpi_rrG(x)
$\sqrt{x^2+y^2}$	sqrtx2y2_rrG(x, y)	cos($\pi \cdot x$)	cospi_rrG(x)
$\sqrt{1+x^2}$	sqrt1px2_rrG(x)	tan($\pi \cdot x$)	tanpi_rrG(x)
$\sqrt{1-x^2}$	sqrt1mx2_rrG(x)	cot_rrG($\pi \cdot x$)	cotpi_rrG(x)
$\sqrt{x^2-1}$	sqrtx2m1_rrG(x)	arcsin(x)	arcsin_rrG(x)
e^x	exp_rrG(x)	arccos(x)	arccos_rrG(x)
$e^x - 1$	expm1_rrG(x)	arctan(x)	arctan_rrG(x)
e^{-x^2}	exp_x2_rrG(x)	arctan2(x, y)	arctan2_rrG(x, y)
2^x	exp2_rrG(x)	arccot(x)	arccot_rrG(x)
10^x	exp10_rrG(x)	sinh(x)	sinh_rrG(x)
x^y	power_rrG(x, y)	cosh(x)	cosh_rrG(x)
$\ln(x)$	ln(x)	tanh(x)	tanh_rrG(x)
$\ln(e \cdot x)$	ln_ex_rrG(x)	coth(x)	coth_rrG(x)
$\ln(1+x)$	ln1p_rrG(x)	arsinh(x)	arsinh_rrG(x)
$\ln(x^2+y^2)/2$	ln_sqrtx2y2_rrG(x, y)	arcosh(x)	arcosh_rrG(x)
$\ln[(1+x)^2+y^2)]/2$	ln_sqrt1px2y2_rrG(x, y)	arcosh($1+x$)	arcosh1p_rrG(x)
$\log_a(x)$	loga_rrG(a, x)	artanh(x)	artanh_rrG(x)
$\log_2(x)$	log2_rrG(x)	arcoth(x)	arcoth_rrG(x)
$\log_{10}(x)$	log10_rrG(x)	diam(x)	diam_rrG(x)
$\log_{10}(1+x)$	log1p_rrG(x)		

Tabelle 4.9: Mathematische Standardfunktionen des Moduls RRGI_ARI;
x, y: Linterval; a: Longreal;

4.3.5 Überladungen des Zuweisungsoperators

Die Wandlungen von *integer/real/Longreal/rrG/Linterval* nach *rrGinterval* bzw. von *rrGinterval* nach *Linterval* werden auch durch Überladungen des Zuweisungsoperators bereitgestellt:

Zuweisung	**Bedeutung**
rrGi := rrGx	rrGi := RRGINTVAL(rrGx); rrGi : rrGinterval; rrGx : rrG
rrGi := Lx	rrGi := RRGINTVAL(Lx); rrGi : rrGinterval; Lx : Longreal
rrGi := x	rrGi := RRGINTVAL(x); rrGi : rrGinterval; x : real
rrGi := Li	rrGi ≡ Li; rrGi : rrGinterval; LI : Linterval
Li := rrGi	rrGi ⊆ Li; rrGi : rrGinterval; LI : Linterval

4.3.6 Ein-/Ausgabeprozeduren

Es stehen die bekannten Prozeduren

> **procedure** read (**var** f: text; **var** a: rrGinterval);
> **procedure** write (**var** f: text; a: rrGinterval);

mit optionalem Fileparameter, beliebig vielen Ein–/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Intervalls $x = [a, b]$ mit a,b : rrG muß in der Form

$$[a, b] \qquad \text{oder in der Form} \qquad a$$

erfolgen. Der zweite Fall dient zur vereinfachten Eingabe eines Punktintervalls $x = [a, a]$.

Die Eingabe eines Wertes a : rrG muß dabei in der Form

{ a.r1,a.r2,a.G } oder in der Form a.r1

erfolgen. Weitere Einzelheiten findet man auf Seite 56.

Die Ausgabe eines Intervalls durch die write–Prozedur erfolgt stets in der Form

$$[a, b]$$

wobei z.B. a : rrG in der Form

{ a.r1,a.r2,a.G }

ausgegeben wird.

4.3.7 Wertebereich einer Funktion

Gegeben sei eine über einem Intervall $\mathbb{D} = [a, b]$ definierte, reellwertige Funktion f

$$f \; : \; \mathbb{D} = [a, b] \longrightarrow \mathbb{R};$$

Der Wertebereich $\mathbb{W}$ von f ist dann definiert durch:

$$\mathbb{W} := \{y \,|\, y = f(x) \wedge x \in [a, b]\}.$$

Wird $f(x)$ für ein rrG–Argument $x \in [a, b]$ mit einem geeigneten Algorithmus ausgewertet, so ist $\widetilde{f}(x) \neq f(x)$ das fehlerbehaftete Rechnerergebnis mit dem relativen Fehler ε_f, definiert durch:

$$\varepsilon_f := \frac{\widetilde{f}(x) - f(x)}{f(x)}; \quad f(x) \neq 0; \qquad |\varepsilon_f| \leq \varepsilon(f), \quad x \in [a, b].$$

Die Fehlerschranke $\varepsilon(f)$ wird als bekannt angesehen. Die PASCAL–XSC–Funktion

function f (x: rrG): rrG;

liefert zum exakten Funktionswert $f(x)$ die rrG–Näherung $\widetilde{f}(x)$.

Gesucht ist eine Intervallfunktion vom Ergebnistyp *rrGinterval*, die zu einem gegebenem Intervall $\mathrm{I} \subset [a, b]$ vom Typ *rrGinterval* eine **Einschließung** des entsprechenden Wertebereichs

$$\mathbb{W}_I := \{y \,|\, y = f(x) \wedge x \in \mathrm{I} \subset [a, b]\}.$$

liefert. Unter bestimmten Voraussetzungen lassen sich die gesuchten Intervallfunktionen mit vordefinierten Funktionen aus dem Modul **rrGi_ari** realisieren:

4.3.7.1 Die Funktion ist monoton

Mit Hilfe der Funktion **RRGINTVAL_f_rrG**(...) aus dem Modul **rrGi_ari** erhält man die gewünschte Einschließung des Wertebereichs:

function **RRGINTVAL_f_rrG** (function f (x: rrG): rrG; I: rrGinterval;
bl: boolean; eps: real): rrGinterval;

Zu einer in einem Intervall $\mathrm{I} = [x_1, x_2]$ monotonen Funktion f berechnet

w := **RRGINTVAL_f_rrG** (f, I, bl, eps)

eine garantierte Einschließung w der Funktionswerte $f(x)$, mit $x \in \mathrm{I} = [x_1, x_2]$:

$$\mathbb{W}_I := \{y \,|\, y = f(x) \;\wedge\; x \in \mathrm{I}\} \subseteq \mathrm{w}\,;$$

Ist f in I monoton wachsend (fallend), so muß bl = $true$ ($false$) gesetzt werden.
Ist $\varepsilon(f)$ die relative Fehlerschranke von f, so muß eps wie folgt berechnet werden:

eps := 1.00001*> $\varepsilon(f)$;

Im nachfolgenden Beispiel wird gezeigt, wie die Exponentialfunktion

function exp (x: rrGinterval): rrGinterval;

aus dem Modul **rrGi_ari** realisiert wurde:

```
function EXP(x: rrGinterval): rrGinterval;
var c : rrGinterval;
begin
  c := rrGintval_f_rrG(EXP,x,TRUE,4.6214E-25);
  if SIGN(c.SUP.r1) = 0 then
  begin { Oberschranke falls Underflow: }
    c.SUP.r1 := 1;  c.SUP.r2 := 0;
    c.SUP.G  := -2171;
  end;
  EXP := c;
end;  { EXP }
```

EXP ist dabei der Name der Exponentialfunktion für rrG–Argumente aus der Tabelle von Seite 58. Mit der Fehlerschranke $\varepsilon(f) = 4.6213\,\text{E}-25$ ergibt sich dann eps zu: $\text{eps} = 4.6214\,\text{E}-25$.

Mit den beiden folgenden Funktionen aus dem Modul rrGi_ari lassen sich Wertebereiche von monotonen Funktionen ganz entsprechend berechnen, wenn das Monotonie–Intervall oder das Argument von f andere Datentypen besitzen:

function **RRGINTVAL_f** (function f (x: Longreal): rrG; I: Linterval;
bl: boolean; eps: real): rrGinterval;

function **RRGINTVAL_frrG** (function f (x: rrG): rrG; I: Linterval;
bl: boolean; eps: real): rrGinterval;

4.3.7.2 Die Funktion besitzt ein relatives Maximum

$f\,:\,[a,b] \longrightarrow \mathbb{R}$ sei eine stetige Funktion und habe in $x_0 \in \,]\,a,b\,[$ ein relatives Maximum. Für $x < x_0$ sei f monoton wachsend, und für $x > x_0$ sei f monoton fallend.
$f(x)$ wird auf dem Rechner durch folgende Funktion realisiert:

function f (x: rrG): rrG;

und liefert zum rrG–Argument $x \in [a,b]$ den i.a. fehlerbehafteten Funktionswert $\widetilde{f}(x) \neq f(x)$.
Die zugehörige relative Fehlerschranke $\varepsilon(f)$ wird für alle $x \in [a,b]$ als bekannt angenommen.

Zu einem beliebig vorgegebenen Intervall I vom Typ *rrGinterval*, mit:

$$\text{I} = [\,\text{I.inf, I.sup}\,] \subseteq [a,b]$$

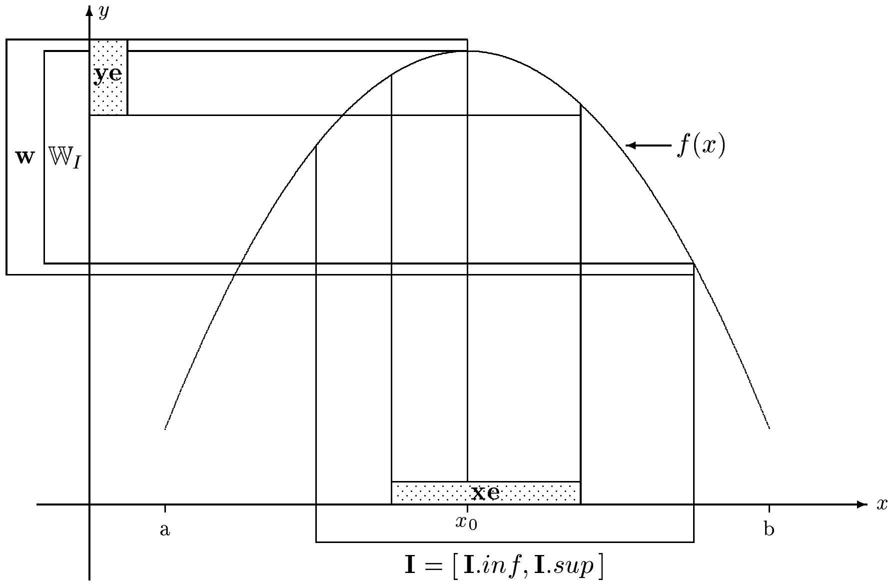


Abbildung 4.1: Einschließung des Wertebereichs $\mathbb{W}_I$ durch das Intervall $\mathbf{w}$

ist gesucht eine Einschließung w = [w.inf, w.sup] vom Typ *rrGinterval* für den Wertebereich:

$$\mathbb{W}_I := \{y \,|\, y = f(x) \ \wedge \ x \in \mathrm{I} = [\mathrm{I.inf, I.sup}]\,\} \subseteq \mathrm{w} = [\mathrm{w.inf, w.sup}]\,;$$

In Abb. 4.1 ist *xe* ein Intervall vom Typ *rrGinterval*, welches die Maximumstelle x_0 einschließt. Das Intervall *ye*, ebenfalls vom Typ *rrGinterval*, ist eine Einschließung aller Funktionswerte $f(x)$, mit $x \in xe$:

$$\{\, y \,|\, y = f(x) \ \wedge \ x \in xe\,\} \subseteq ye\,;$$

Ist die relative Fehlerschranke $\varepsilon(f)$ bekannt, so lassen sich die Einschließungen *xe*,*ye* mit dem Programm **MIN_26** bestimmen, wenn man dort $-f(x)$ an Stelle von $f(x)$ benutzt, da **MIN_26** die genannten Einschließungen nur unter der Voraussetzung berechnet, daß die betrachtete Funktion ein relatives **Minimum** besitzt.

Mit Hilfe der Funktion

function **INCL_MAX** (function f (x: rrG): rrG; eps: real;
xe, ye, I: rrGinterval): rrGinterval;

die durch **rrGi_ari** bereitgestellt wird, erhält man dann die Einschließung *w* durch den Funktionsaufruf:

w := **INCL_MAX** (f, eps, xe, ye, I);

wobei eps durch eps := 1.00001*> $\varepsilon(f)$ zu berechnen ist.

4.3.7.3 Die Funktion besitzt ein relatives Minimum

$f\,:\,[a,b] \longrightarrow \mathbb{R}$ sei eine stetige Funktion und habe in $x_0 \in]\,a,b\,[$ ein relatives Minimum. Für $x < x_0$ sei f monoton fallend, und für $x > x_0$ sei f monoton wachsend.

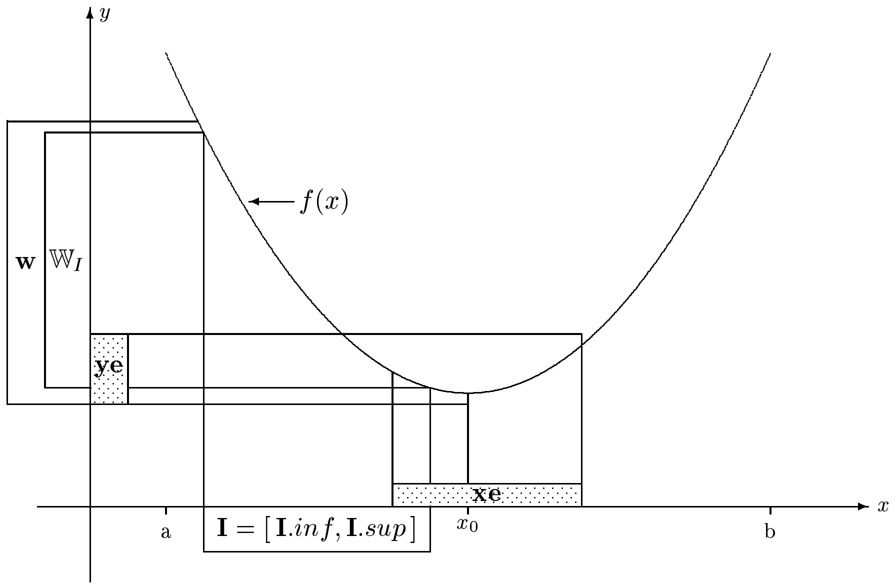


Abbildung 4.2: Einschließung des Wertebereichs $\mathbb{W}_I$ durch das Intervall **w**

$f(x)$ wird auf dem Rechner durch folgende Funktion realisiert:

function f (x: rrG): rrG;

und liefert zum rrG–Argument $x \in [a,b]$ den i.a. fehlerbehafteten Funktionswert $\widetilde{f}(x) \neq f(x)$.
Die zugehörige relative Fehlerschranke $\varepsilon(f)$ wird für alle $x \in [a,b]$ als bekannt angenommen.

Zu einem beliebig vorgegebenen Intervall I vom Typ *rrGinterval*, mit:

$$\mathrm{I} = [\,\mathrm{I.inf},\, \mathrm{I.sup}\,] \subseteq [a,b]$$

ist gesucht eine Einschließung $\mathrm{w} = [\mathrm{w.inf},\, \mathrm{w.sup}]$ vom Typ *rrGinterval* für den Wertebereich:

$$\mathbb{W}_I := \{y \,|\, y = f(x) \;\wedge\; x \in \mathrm{I} = [\mathrm{I.inf},\, \mathrm{I.sup}]\,\} \subseteq \mathrm{w} = [\mathrm{w.inf},\, \mathrm{w.sup}]\,;$$

In Abb. 4.2 ist *xe* ein Intervall vom Typ *rrGinterval*, welches die Minimumstelle x_0 einschließt. Das Intervall *ye*, ebenfalls vom Typ *rrGinterval*, ist eine

Einschließung aller Funktionswerte $f(x)$, mit $x \in xe$:

$$\{\, y \,|\, y = f(x) \;\wedge\; x \in xe \,\} \subseteq ye\,;$$

Ist die relative Fehlerschranke $\varepsilon(f)$ bekannt, so lassen sich die Einschließungen *xe,ye* mit dem Programm **MIN_26** bestimmen.

Mit Hilfe der Funktion

function **INCL_MAX** (function f (x: rrG): rrG; eps: real;
xe, ye, I: rrGinterval): rrGinterval;

aus dem Modul **rrGi_ari** erhält man dann die Einschließung *w* durch folgendes Programmbeispiel:

```
program test(input,output);
use rrGi_ari;
var xe,ye,I,w : rrGinterval;    eps: real;

  function f(x: rrG): rrG;
  begin   { Definition von  f;     x > 1 }
    f := exp(x) / ln(x);
  end;    { Rel. Fehlerschr.: 5.4457E-24 }

  function fI(xI: rrGinterval): rrGinterval;
  { fI liefert zu gegebenem  Intervall I }
  { die Einschliessung  w von W_I        }
    function fn(x: rrG): rrG;
    begin  fn := -f(x)  end;
  begin  fI := -INCL_MAX(fn,eps,xe,-ye,xI)  end;

begin { Haupt }
  eps := 5.4458E-24;   { = 1.00001 *> 5.4457E-24 }
  {* Ergebnisse des Programms  MIN_26 :         *}
  xe.INF.r1 := 1.763222834346;
  xe.INF.r2 := 3.793862611114E-13;  xe.INF.G := 0;
  xe.SUP.r1 := 1.763222834357;
  xe.SUP.r2 := 2.933226823867E-13;  xe.SUP.G := 0;

  ye.INF.r1 := 1.028170523089E+1;
  ye.INF.r2 := -2.754504467672E-12; ye.INF.G := 0;
  ye.SUP.r1 := 1.028170523089E+1;
  ye.SUP.r2 := -2.754504466670E-12; ye.SUP.G := 0;

  I.inf := 1.6; I.sup := 1.8; {Eingangsintervall }
  w := fI(I);   { w: Wertebereichseinschliessung }
end. { f(x) > 1.028170523089E+1 - 2.754504467672E-12 }
```

4.3.7.4 Die Funktion besitzt ein realtives Maximum und Minimum

$f\,:\,[a,b] \longrightarrow \mathbb{R}$ sei eine stetige Funktion und habe in $x_0 \in \,]\,a,b\,[$ ein relatives Maximum und in $x_1 > x_0$ ein relatives Minimum. Vor, zwischen und nach den relativen Extrema sei $f(x)$ jeweils monoton.

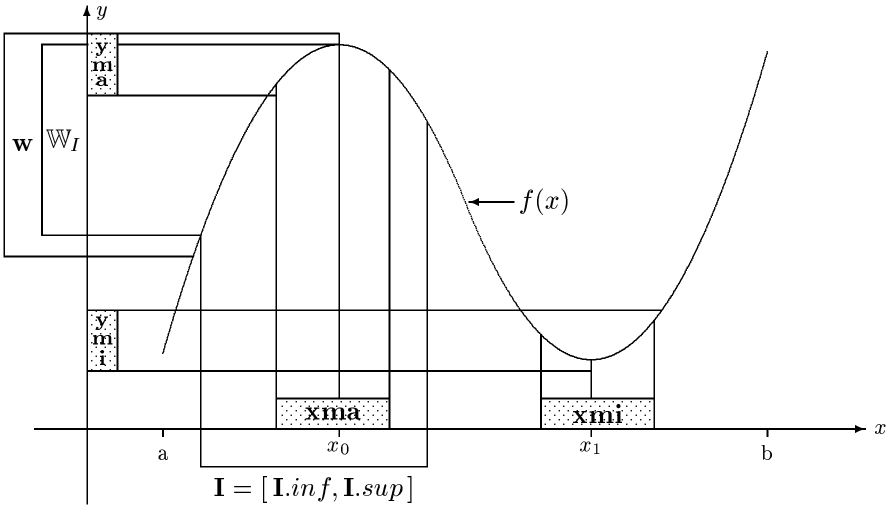


Abbildung 4.3: Einschließung des Wertebereichs $\mathbb{W}_I$ durch das Intervall **w**

$f(x)$ wird auf dem Rechner durch folgende Funktion realisiert:

function f (x: rrG): rrG;

und liefert zum rrG–Argument $x \in [a,b]$ den i.a. fehlerbehafteten Funktionswert $\widetilde{f}(x) \neq f(x)$.
Die zugehörige relative Fehlerschranke $\varepsilon(f)$ wird für alle $x \in [a,b]$ als bekannt angenommen.

Zu einem beliebig vorgegebenen Intervall I vom Typ *rrGinterval*, mit:

$$\mathrm{I} = [\,\mathrm{I.inf},\,\mathrm{I.sup}\,] \subseteq [a,b]$$

ist gesucht eine Einschließung w = [w.inf, w.sup] vom Typ *rrGinterval* für den Wertebereich:

$$\mathbb{W}_I := \{y \,|\, y = f(x) \;\wedge\; x \in \mathrm{I} = [\mathrm{I.inf},\,\mathrm{I.sup}]\,\} \subseteq \mathrm{w} = [\mathrm{w.inf},\,\mathrm{w.sup}]\,;$$

In Abb. 4.3 ist *xma* ein Intervall vom Typ *rrGinterval*, welches die Maximumstelle x_0 einschließt. Das Intervall *yma*, ebenfalls vom Typ *rrGinterval*, ist eine Einschließung aller Funktionswerte $f(x)$, mit $x \in xma$:

$$\{\,y \,|\, y = f(x) \;\wedge\; x \in xma\,\} \subseteq yma\,;$$

Entsprechend ist *xmi* ein Intervall vom Typ *rrGinterval*, welches die Minimumstelle x_1 einschließt. Das Intervall *ymi*, ebenfalls vom Typ *rrGinterval*, ist eine Einschließung aller Funktionswerte $f(x)$, mit $x \in xmi$:

$$\{ y \,|\, y = f(x) \;\wedge\; x \in xmi \} \subseteq ymi\,;$$

Ist die relative Fehlerschranke $\varepsilon(f)$ bekannt, so lassen sich die Einschließungen *xma, yma* bzw. *xmi, ymi* mit dem Programm **MIN_26** bestimmen.

Mit Hilfe der Funktion

function **INCL_MAMI** (function f (x: rrG): rrG; eps: real;
xma, yma, xmi, ymi, I: rrGinterval): rrGinterval;

aus dem Modul **rrGi_ari** erhält man dann die Einschließung *w* durch folgenden Funktionsaufruf:

w := **INCL_MAMI** (f, eps, xma, yma, xmi, ymi, I);

wobei eps wie folgt zu berechnen ist: eps := 1.00001*> $\varepsilon(f)$;

4.3.7.5 Die Funktion besitzt ein realtives Minimum und Maximum

$f\,:\,[a,b] \longrightarrow \mathbb{R}$ sei eine stetige Funktion und habe in $x_0 \in]\,a, b\,[$ ein relatives Minimum und in $x_1 > x_0$ ein relatives Maximum.
Vor, zwischen und nach den relativen Extrema sei f jeweils monoton.

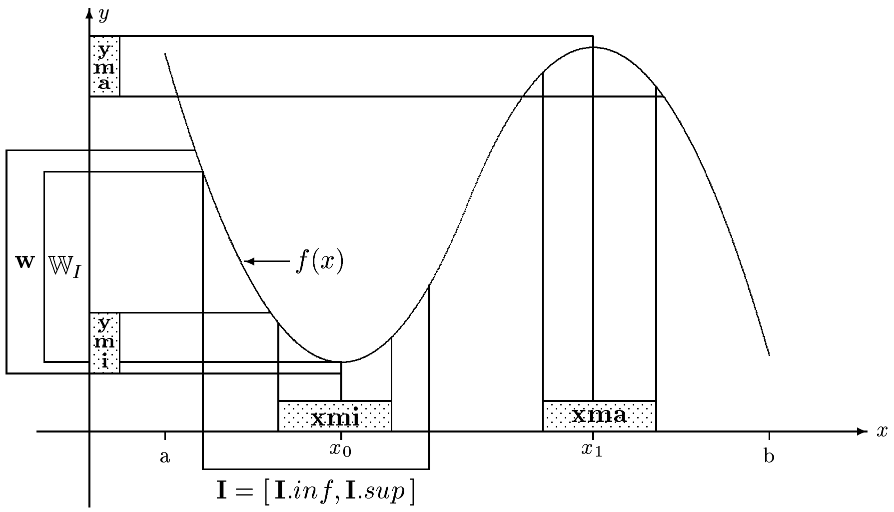


Abbildung 4.4: Einschließung des Wertebereichs $\mathbb{W}_I$ durch das Intervall **w**

$f(x)$ wird auf dem Rechner durch folgende Funktion realisiert:

function f (x: rrG): rrG;

und liefert zum rrG–Argument $x \in [a, b]$ den i.a. fehlerbehafteten Funktionswert $\widetilde{f}(x) \neq f(x)$.
Die zugehörige relative Fehlerschranke $\varepsilon(f)$ wird für alle $x \in [a, b]$ als bekannt angenommen.

Zu einem beliebig vorgegebenen Intervall I vom Typ *rrGinterval*, mit:

$$\mathrm{I} = [\,\mathrm{I.inf},\ \mathrm{I.sup}\,] \subseteq [a, b]$$

ist gesucht eine Einschließung w = [w.inf, w.sup] vom Typ *rrGinterval* für den Wertebereich:

$$\mathbb{W}_I := \{y \,|\, y = f(x) \ \wedge \ x \in \mathrm{I} = [\mathrm{I.inf},\ \mathrm{I.sup}]\,\} \subseteq \mathrm{w} = [\mathrm{w.inf},\ \mathrm{w.sup}]\,;$$

In Abb. 4.4 ist *xmi* ein Intervall vom Typ *rrGinterval*, welches die Minimumstelle x_0 einschließt. Das Intervall *ymi*, ebenfalls vom Typ *rrGinterval*, ist eine Einschließung aller Funktionswerte $f(x)$, mit $x \in xmi$:

$$\{\, y \,|\, y = f(x) \ \wedge \ x \in xmi\,\} \subseteq ymi\,;$$

Entsprechend ist *xma* ein Intervall vom Typ *rrGinterval*, welches die Maximumstelle x_1 einschließt. Das Intervall *yma*, ebenfalls vom Typ *rrGinterval*, ist eine Einschließung aller Funktionswerte $f(x)$, mit $x \in xma$:

$$\{\, y \,|\, y = f(x) \ \wedge \ x \in xma\,\} \subseteq yma\,;$$

Ist die relative Fehlerschranke $\varepsilon(f)$ bekannt, so lassen sich die Einschließungen *xmi, ymi* bzw. *xma, yma* mit dem Programm **MIN_26** bestimmen.

Mit Hilfe der Funktion

function **INCL_MAMI** (function f (x: rrG): rrG; eps: real;
xma, yma, xmi, ymi, I: rrGinterval): rrGinterval;

aus dem Modul **rrGi_ari** erhält man dann die Einschließung *w* durch folgendes Programmbeispiel:

```
program test (input,output);
use rrGi_ari;
var xmi, ymi, xma, yma, I, w : rrGinterval;
    eps : real;

function f (x: rrG): rrG;
begin
   f := ... { Definition von f }
end;
```

```
function fI (I: rrGinterval): rrGinterval;
   { fI liefert zu gegebenem Intervall I die Einschließung w von 𝕎_I }
   function fn (x: rrG): rrG;
   begin fn := −f(x) end;
begin
   fI := −INCL_MAMI (fn, eps, xmi, −ymi, xma, −yma, I);
end;

begin { Haupt }
   eps := ... ; { eps := 1.00001*> ε(f)}
   xmi := rrGintval (... , ...);
   ymi := rrGintval (... , ...);
   xma := rrGintval (... , ...);
   yma := rrGintval (... , ...);
   I := rrGintval (... , ...);
   w := fI(I);    { w liefert eine Einschließung von 𝕎_I }
end.
```

4.3.7.6 Die Funktion besitzt mehr als zwei Extrema

$f\ :\ [a,b] \longrightarrow \mathbb{R}$ sei eine reellwertige, stetige Funktion und habe in $x_0 \in]\, a,b\, [$ nach Abb. 4.5 z.B. drei relative Extrema.

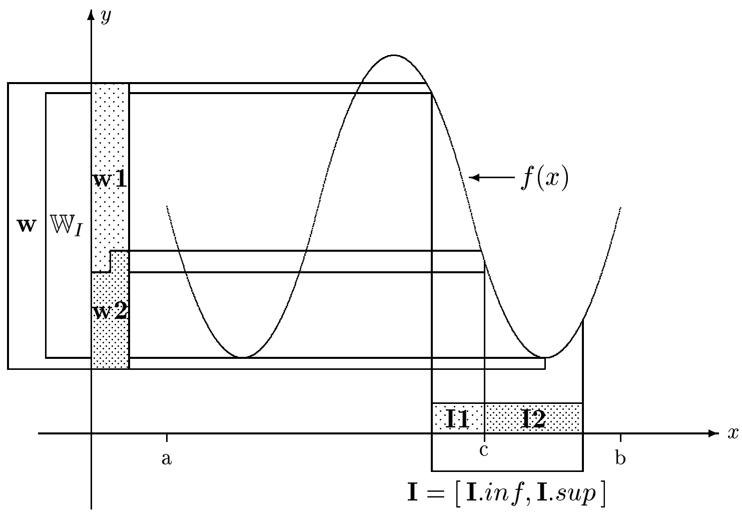


Abbildung 4.5: Funktion mit drei relativen Extrema

$f(x)$ wird auf dem Rechner durch folgende Funktion realisiert:

```
function f (x: rrG): rrG;
```

und liefert zum rrG–Argument $x \in [a, b]$ den i.a. fehlerbehafteten Funktionswert $\widetilde{f}(x) \neq f(x)$.
Die zugehörige relative Fehlerschranke $\varepsilon(f)$ wird für alle $x \in [a, b]$ als bekannt angenommen.

Zu einem beliebig vorgegebenen Intervall I vom Typ *rrGinterval*, mit:

$$\mathrm{I} = [\,\mathrm{I.inf},\ \mathrm{I.sup}\,] \subseteq [a, b]$$

ist gesucht eine Einschließung $\mathrm{w} = [\mathrm{w.inf},\ \mathrm{w.sup}]$ vom Typ *rrGinterval* für den Wertebereich:

$$\mathbb{W}_I := \{y \mid y = f(x) \ \wedge \ x \in \mathrm{I} = [\mathrm{I.inf},\ \mathrm{I.sup}]\,\} \subseteq \mathrm{w}\,;$$

Zur Lösung unterteilt man zunächst $[a, b]$ durch $c \in [a, b]$ in die beiden Teilintervalle $[a, c]$ und $[c, b]$. Mit den Schnittintervallen

$$\mathrm{I1} := \mathrm{I} \cap [a, c] \quad \text{und} \quad \mathrm{I2} := \mathrm{I} \cap [c, b]$$

berechnet man dann entsprechend den Abbildungen 4.4 bzw. 4.2 die jeweiligen Einschließungen w1 und w2. Die konvexe Hülle von w1 und w2 ist dann die gesuchte Einschließung w:

$$\mathrm{w} := \mathrm{w1} +\!* \,\mathrm{w2} \supseteq \mathrm{I}$$

Bei mehr als drei Extrema unterteilt man $[a, b]$ so in Teilintervalle, daß jedes höchstens zwei Extrema enthält und verfährt dann analog zum obigen Beispiel.

Bei bekannter Fehlerschranke $\varepsilon(f)$ lassen sich so für alle 'gutmütigen' Funktionen zu einem beliebigen Intervall–Argument I eine Einschließung w des entsprechenden Wertebereichs $\mathbb{W}_I$ bestimmen, wenn man vorher alle Einschließungen der relativen Extrema mit dem Programm **MIN_26** berechnet hat.

Kapitel 5

Komplexe Arithmetik

Mit dem Modul **C_ARI** werden die für das Rechnen mit komplexen Zahlen notwendigen Operatoren, Funktionen und Prozeduren bezüglich des Datentyps **real** bereitgestellt.

Das Modul **LC_ARI** ermöglicht zusätzlich komplexe Rechnungen auf Basis der Typen **real, Longreal, dotprecision** und **rrG**.

5.1 Das Modul C_ARI

Für Rechnungen mit komplexen Zahlen wird der Datentyp *complex* entsprechend der Definition

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

im Sprachkern von PASCAL–XSC bereitgestellt. Für $z = x + i \cdot y, \quad i = \sqrt{-1}$ und z : *complex* gilt dann:

z.re = x   und   z.im = y;

5.1.1 Operatoren

Sämtliche in diesem Modul vordefinierten arithmetischen Operatoren liefern den Ergebnistyp *complex*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zu Verfügung. Dabei werden die Rundungen jeweils auf Real- **und** Imaginärteil angewandt. Die Ergebnisse der Operationen $+, -, *$ sind **maximalgenau**, d.h. das exakte Ergebnis wird stets zur nächsten Maschinenzahl gerundet. Das Ergebnis der komplexen Division ist jedoch nur **hochgenau**, d.h. zwischen dem exakten Quotienten und dem Maschinenergebnis liegt bzgl. Real- und Imaginärteil **keine** weitere Rasterzahl.
Das Beispiel $c := (2.00\ldots003 - i \cdot 10^{-13})/(2 + i \cdot 10^{-13})$ liefert z.B. für den Realteil die nur hochgenaue Maschinennäherung $\widetilde{\mathrm{Re}}(c) = 1.00\ldots002$ im Gegensatz zur maximalgenauen Näherung: 1.00000000000**1**.

Bezeichnen wir mit $\varepsilon(G)$ die relative Fehlerschranke der Operationen $+, -, *$ und mit $\varepsilon(D)$ die relative Fehlerschranke der Division und sind $\hat{\varepsilon}(G), \hat{\varepsilon}(D)$ die entsprechenden Fehlerschranken für die gerichteten Rundungen, so gelten für den normalisierten bzw. denormalisierten real–Bereich die in folgender Tabelle angegebenen relativen Fehlerschranken:

Oper.	**Normal.**	**Denormal.**	**Rel. Fehlerschranke**
$+/-$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G) = 5 \cdot 10^{-13};\quad \hat{\varepsilon}(G) = 1 \cdot 10^{-12}$
$*$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$> \varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G) = 5 \cdot 10^{-13};\quad \hat{\varepsilon}(G) = 1 \cdot 10^{-12}$
$/$	$\varepsilon(D);\ \hat{\varepsilon}(D)$	$> \varepsilon(D);\ \hat{\varepsilon}(D)$	$\varepsilon(D) = 1 \cdot 10^{-12};\quad \hat{\varepsilon}(D) = 1.5 \cdot 10^{-12}$

Liegen die Ergebnisse von Multiplikation oder Division also im denormalisierten Bereich (3.1), so kann der relative Betragsfehler die Fehlerschranken aus dem normalisierten Bereich überschreiten, vergl. dazu Seite 12. Es ist daher sinnvoll, für die komplexe Multiplikation und Division Funktionen bereitzustellen, die für Real- und Imaginärteilergebnisse aus dem denormalisierten real–Bereich überprüfen, ob die Beträge der aufgetretenen relativen Fehler die Fehlerschranken aus dem normalisierten Bereich übertreffen:

1. **function** MULNEXT_TEST(var A,B: complex; var error: boolean): complex;
   Berechnet A * B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $> 0.5 \cdot 10^{-12}$;

2. **function** MULUP_TEST(var A,B: complex; var error: boolean): complex;
   Berechnet A *> B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $\geq 1 \cdot 10^{-12}$;

3. **function** MULDOWN_TEST(var A,B:complex; var error:boolean): complex;
   Berechnet A *< B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $\geq 1 \cdot 10^{-12}$;

4. **function** DIVNEXT_TEST(var A,B: complex; var error: boolean): complex;
   Berechnet A / B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $> 1 \cdot 10^{-12}$;

5. **function** DIVUP_TEST(var A,B: complex; var error: boolean): complex;
   Berechnet A /> B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $> 1.5 \cdot 10^{-12}$;

6. **function** DIVDOWN_TEST(var A,B: complex; var error: boolean): complex;
   Berechnet A /< B; error = TRUE $\rightsquigarrow$ |rel. Fehler| $> 1.5 \cdot 10^{-12}$;

Die Vergleichsoperatoren = <> < <= > >= beziehen sich auf Real– und Imaginärteil. Für a,b vom Typ *complex* gilt demnach:

$$\text{a} <= \text{b} \iff (\text{a.re} <= \text{b.re}) \ \ \textbf{and} \ \ (\text{a.im} <= \text{b.im})$$

Entsprechende Vergleiche auch mit einem *integer*– oder *real*–Operanden sind zulässig.

rechter Operand	integer real	complex
linker Operand		
monadisch		$+, -$
integer real		$\circ$ $\vee$
complex	$\circ$ $\vee$	$\circ$ $\vee$

Tabelle 5.1: Die Operatoren des Moduls C_ARI

$$\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>, /, /<, />\}$$

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

5.1.2 Transferfunktionen

Zur Wandlung zwischen den Typen *integer/real* und *complex* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
compl (r1,r2)	*complex*	$z = r1 + i \cdot r2; \quad i = \sqrt{-1}$
compl (r)	*complex*	$z = r1 + i \cdot 0; \quad i = \sqrt{-1}$
re (z)	*real*	Realteil von z
im (z)	*real*	Imaginärteil von z

Tabelle 5.2: Transferfunktionen des Moduls C_ARI
r, r1, r2 = *real*–Ausdruck; z = *complex*-Ausdruck

5.1.3 Überladungen des Zuweisungsoperators

Die Wandlung von *real/integer* nach *complex* wird auch in Form einer überladenen Zuweisung bereitgestellt:

Zuweisung	Bedeutung
z := r	z := compl (r)

z = *complex*–**Variable**; r = *real*–Ausdruck

5.1.4 Standardfunktionen

Es stehen die folgenden Standardfunktionen zur Verfügung:

Funktion	Bedeutung	Aufruf	Ergebnistyp
$z^2 = z \cdot z$	Quadrat	sqr (z)	complex
$\sqrt{z}$	Quadratwurzel (Realteil ≥ 0)	sqrt (z)	complex
$\sqrt{z}$	Quadratwurzel (Realteil ≥ 0)	sqrt_u (z)	complex
$\sqrt{z}$	Quadratwurzel	sqrt_negim (z)	complex
e^z	Exponentialfunktion	exp (z)	complex
$\ln(z)$	Natürlicher Logarithmus	ln (z)	complex
$\ln(1+z)$	Natürlicher Logarithmus	ln1p (z)	complex
$\sin(z)$	Sinus	sin (z)	complex
$\cos(z)$	Kosinus	cos (z)	complex
$\tan(z)$	Tangens	tan (z)	complex
$\cot(z)$	Kotangens	cot (z)	complex
$\sinh(z)$	Hyperbolischer Sinus	sinh (z)	complex
$\cosh(z)$	Hyperbolischer Kosinus	cosh (z)	complex
$\tanh(z)$	Hyperbolischer Tangens	tanh (z)	complex
$\coth(z)$	Hyperbolischer Kotangens	coth (z)	complex
$\bar{z} = x - i \cdot y$	Konjugation von $z = x + i \cdot y$	conj (z)	complex
φ	Argument von $z = r \cdot e^{i\varphi}$	arg (z)	real
φ	Argument von $1 + z = r \cdot e^{i\varphi}$	arg1p (z)	real
$r = \sqrt{x^2 + y^2}$	Absolutbetrag von $z = r \cdot e^{i\varphi}$	abs (z)	real

Tabelle 5.3: Standardfunktionen des Moduls C_ARI, z: complex

Anmerkungen zu den Funktionen: sqrt(z), sqrt_u(z), sqrt_negim, ln(z), ln1p(z):

- **sqrt(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.
- **sqrt_u(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur unteren Halbebene zählt.
- **sqrt_negim(z)** Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse zur linken Halbebene zählt.
- **ln(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.

- **ln1p(z)** Der Verzweigungsschnitt ist der negative Teil der reellen Achse von $-\infty$ bis -1, wobei diese Achse zur oberen Halbebene zählt.

Für alle Funktionen aus Tabelle 5.3 ist der Betrag des relativen Fehlers kleiner als $5.00000006 \cdot 10^{-13}$, so daß alle Funktionen hochgenau sind, d.h. das Maschinenergebnis unterscheidet sich vom optimal gerundeten Funktionsergebnis um maximal eine Einheit in der letzten Mantissenstelle. Diese Abweichung von 1 ulp tritt allerdings nur in **sehr seltenen Ausnahmefällen** auf!

Fallen Real– oder Imaginärteil in den denormalisierten Bereich (3.1), d.h.

$$0 \leq |\mathrm{Re}(f(z))|, |\mathrm{Im}(f(z))| < 10^{-255} = \mathrm{MIN\_REAL},$$

und werden dadurch hintere, von Null verschiedene Ziffern der Ergebnismantisse gleich Null gesetzt oder entsteht Underflow, so daß die Funktionsergebnisse dadurch nicht mehr hochgenau sind, so erfolgt die Fehlermeldung:

rel. error exceeds error bound

mit anschließendem Programmabbruch.

Die noch unvollständige Liste der komplexen Standardfunktionen aus Tabelle 5.3 wird in folgenden Updates erweitert.

5.1.5 Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

procedure read (**var** f: text; **var** a: complex);
procedure write (**var** f: text; a: complex);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe einer komplexen Zahl $z = x + i \cdot y$ muß in der Form

$$(x, y) \qquad \text{oder in der Form} \qquad x$$

erfolgen. Im zweiten Fall wird der Imaginärteil y automatisch auf 0 gesetzt. x und y sind dabei *real*-Konstanten. Die Ausgabe einer komplexen Zahl erfolgt stets in der Form

$$(x, y)$$

mit dem Standardformat für die *real*-Größen x und y aus Tabelle 3.4.

5.2 Das Modul LC_ARI

5.2.1 Der Typ Lcomplex

Für allgemeine Gleitkommarechnungen mit komplexen Zahlen wird neben dem Typ *complex* zusätzlich der Datentyp *Lcomplex* entsprechend der Definition

type Lcomplex = **record** Re,Im : Longreal **end**;

bereitgestellt. Für $z = x + i \cdot y$, $i = \sqrt{-1}$ und z : *Lcomplex* gilt dann:

z.Re = x und z.Im = y;

5.2.1.1 Operatoren

Wir betrachten zunächst alle in diesem Modul vordefinierten arithmetischen Operatoren mit dem Ergebnistyp *Lcomplex*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zu Verfügung. Dabei werden die Rundungen jeweils auf Real- **und** Imaginärteil angewandt.

rechter Operand	integer real	Longreal	complex	Lcomplex
linker Operand				
monadisch				$+, -$
integer real				$\circ$ $\vee$
Longreal			$\circ$ $\vee$	$\circ$ $\vee$
complex		$\circ$ $\vee$		$\circ$ $\vee$
Lcomplex	$\circ$ $\vee$	$\circ$ $\vee$	$\circ$ $\vee$	$\circ$ $\vee$

Tabelle 5.4: Operatoren des Moduls LC_ARI mit Ergebnistyp *Lcomplex, boolean*

$$\circ \in \{+,\ +<,\ +>,\ -,\ -<,\ ->,\ *,\ *<,\ *>,\ /,\ /<,\ />\}$$

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

Die Ergebnisse der Operationen $+, -, *$ sind **maximalgenau**, d.h. das exakte Ergebnis wird stets zur nächsten Maschinenzahl gerundet. Das Ergebnis der komplexen Division ist im Gegensatz dazu jedoch nur **hochgenau**, d.h. zwischen dem exakten Quotienten und dem Maschinenergebnis liegt bezüglich Real- und Imaginärteil **keine** weitere Rasterzahl.

Das Beispiel $c := (2.00\ldots003 - i\cdot10^{-21})/(2+i\cdot10^{-21})$ liefert z.B. für den Realteil die nur hochgenaue Maschinennäherung $\widetilde{\mathrm{Re}}(c) = 1.00\ldots002$ im Gegensatz zur maximalgenauen Näherung: $1.0000000000000000000\mathbf{1}$.

Bezeichnen wir mit $\varepsilon(G)$ die relative Fehlerschranke der Operationen $+,-,*$ und mit $\varepsilon(D)$ die relative Fehlerschranke der Division und sind $\hat{\varepsilon}(G), \hat{\varepsilon}(D)$ die entsprechenden Fehlerschranken für die gerichteten Rundungen, so gelten für den normalisierten bzw. denormalisierten real–Bereich die in folgender Tabelle angegebenen relativen Fehlerschranken:

Oper.	**Normal.**	**Denormal.**	**Rel. Fehlerschranke**
$+/-$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G) = 5\cdot10^{-21};\quad \hat{\varepsilon}(G) = 1\cdot10^{-20}$
$*$	$\varepsilon(G);\ \hat{\varepsilon}(G)$	$>\varepsilon(G);\ \hat{\varepsilon}(G)$	$\varepsilon(G) = 5\cdot10^{-21};\quad \hat{\varepsilon}(G) = 1\cdot10^{-20}$
$/$	$\varepsilon(D);\ \hat{\varepsilon}(D)$	$>\varepsilon(D);\ \hat{\varepsilon}(D)$	$\varepsilon(D) = 1\cdot10^{-20};\quad \hat{\varepsilon}(D) = 1.5\cdot10^{-20}$

Liegen die Real- bzw. Imaginärteil–Ergebnisse von Multiplikation oder Division also im denormalisierten Bereich, so kann der relative Betragsfehler die Fehlerschranken aus dem normalisierten Bereich überschreiten, vergl. dazu Seite 23. Es ist daher sinnvoll, für die komplexe Multiplikation und Division Funktionen bereitzustellen, die für Real- und Imaginärteilergebnisse aus dem denormalisierten real–Bereich überprüfen, ob die Beträge der aufgetretenen relativen Fehler die Fehlerschranken aus dem normalisierten Bereich übertreffen:

1. **function** MULNEXT_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A $*$ B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| > 0.5\cdot10^{-20}$;

2. **function** MULUP_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A $*>$ B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| \geq 1\cdot10^{-20}$;

3. **function** MULDOWN_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A $*<$ B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| \geq 1\cdot10^{-20}$;

4. **function** DIVNEXT_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A / B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| > 1\cdot10^{-20}$;

5. **function** DIVUP_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A $/>$ B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| > 1.5\cdot10^{-20}$;

6. **function** DIVDOWN_TEST (var A,B : Lcomplex;
   var error : boolean) : Lcomplex;
   Berechnet A $/<$ B; error = TRUE $\rightsquigarrow$ $|$rel. Fehler$| > 1.5\cdot10^{-20}$;

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ beziehen sich auf Real– und Imaginärteil. Für a,b vom Typ *Lcomplex* gilt demnach:

$$\text{a} \;<=\; \text{b} \iff (\text{a.Re} \;<=\; \text{b.Re}) \quad \textbf{and} \quad (\text{a.Im} \;<=\; \text{b.Im})$$

Entsprechende Vergleiche auch mit einem *integer/real*- oder *Longreal*-Operanden sind zulässig.

5.2.1.2 Transferfunktionen

Zur Wandlung zwischen den Typen *integer/real/Longreal/complex* und dem Typ *Lcomplex* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
LCOMPL (x1,x2)	*Lcomplex*	$z = x1 + i \cdot x2; \quad i = \sqrt{-1}$
LCOMPL (x)	*Lcomplex*	$z = x + i \cdot 0; \quad i = \sqrt{-1}$
LONG (c)	*Lcomplex*	$z = \text{LONG}(c.Re) + i \cdot \text{LONG}(c.Im)$
RE (z)	*Longreal*	Realteil von z
IM (z)	*Longreal*	Imaginärteil von z
SHORT (z)	*complex*	Rundung zum nächsten c
SHORTUP (z)	*complex*	Rundung zum nächstgrößeren c
SHORTDOWN (z)	*complex*	Rundung zum nächstkleineren c
short_test(z,error), shortup_test(z,error), shortdown_test(z,error) Rundgn. wie oben; error = true ⇔ rel. Fehler $> \mathbf{5 \cdot 10^{-13}}$ bzw. $\geq 10^{-12}$		

Tabelle 5.5: Transferfunktionen des Moduls LC_ARI

x, x1, x2 = *integer/real/Longreal*–Ausdruck; c = *complex*-Ausdruck;
z = *Lcomplex-Ausdruck*; error = *boolean*-Variable;

5.2.1.3 Überladungen des Zuweisungsoperators

Die Wandlung von *integer/real/Longreal/complex* nach *Lcomplex* wird auch in Form einer überladenen Zuweisung bereitgestellt:

Zuweisung	Bedeutung
z := x	z := LCOMPL (x)
z := c	z := LONG (c)

z = *Lcomplex*-**Variable**; x = *integer/real/Longreal*-Ausdruck;
c = *complex*-Ausdruck

5.2.1.4 Standardfunktionen

Es stehen die folgenden Standardfunktionen zur Verfügung:

Aufruf	Ergebnistyp	Funktion	$\varepsilon(\mathrm{Re}\{f(z)\})$	$\varepsilon(\mathrm{Im}\{f(z)\})$
sqr (z)	Lcomplex	$z^2 = z \cdot z$	$5.0002 \cdot 10^{-21}$	$5.0002 \cdot 10^{-21}$
sqrt (z)	Lcomplex	$\sqrt{z}$	$5.0045 \cdot 10^{-21}$	$5.0045 \cdot 10^{-21}$
sqrt_u (z)	Lcomplex	$\sqrt{z}$	$5.0045 \cdot 10^{-21}$	$5.0045 \cdot 10^{-21}$
sqrt_negim (z)	Lcomplex	$\sqrt{z}$	$5.0045 \cdot 10^{-21}$	$5.0045 \cdot 10^{-21}$
exp (z)	Lcomplex	e^z	$5.0016 \cdot 10^{-21}$	$5.0016 \cdot 10^{-21}$
ln (z)	Lcomplex	$\ln(z)$	$5.0053 \cdot 10^{-21}$	$5.0054 \cdot 10^{-21}$
ln1p (z)	Lcomplex	$\ln(1+z)$	$5.0030 \cdot 10^{-21}$	$5.0059 \cdot 10^{-21}$
sin (z)	Lcomplex	$\sin(z)$	$5.0028 \cdot 10^{-21}$	$5.0038 \cdot 10^{-21}$
cos (z)	Lcomplex	$\cos(z)$	$5.0028 \cdot 10^{-21}$	$5.0038 \cdot 10^{-21}$
tan (z)	Lcomplex	$\tan(z)$	$5.0098 \cdot 10^{-21}$	$5.0120 \cdot 10^{-21}$
cot (z)	Lcomplex	$\cot(z)$	$5.0098 \cdot 10^{-21}$	$5.0120 \cdot 10^{-21}$
sinh (z)	Lcomplex	$\sinh(z)$	$5.0038 \cdot 10^{-21}$	$5.0028 \cdot 10^{-21}$
cosh (z)	Lcomplex	$\cosh(z)$	$5.0028 \cdot 10^{-21}$	$5.0038 \cdot 10^{-21}$
tanh (z)	Lcomplex	$\tanh(z)$	$5.0120 \cdot 10^{-21}$	$5.0098 \cdot 10^{-21}$
coth (z)	Lcomplex	$\coth(z)$	$5.0120 \cdot 10^{-21}$	$5.0098 \cdot 10^{-21}$
conj (z)	Lcomplex	$\bar{z} = x - i \cdot y$	0.0	0.0
arg (z)	Longreal	$\varphi\,; \quad z = r \cdot e^{i \cdot \varphi}\,;$	$5.0054 \cdot 10^{-21}$	
arg1p (z)	Longreal	$\varphi\,; \quad 1+z = r \cdot e^{i \cdot \varphi}\,;$	$5.0059 \cdot 10^{-21}$	
abs (z)	Longreal	$r = \sqrt{x^2+y^2}\,;$	$5.0013 \cdot 10^{-21}$	

Tabelle 5.6: Standardfunktionen des Moduls LC_ARI, z: Lcomplex

Anmerkungen zu den Funktionen: sqrt(z), sqrt_u(z), sqrt_negim, ln(z), ln1p(z):

- **sqrt(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.
- **sqrt_u(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur unteren Halbebene zählt.
- **sqrt_negim(z)** Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse zur linken Halbebene zählt.
- **ln(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.
- **ln1p(z)** Der Verzweigungsschnitt ist der negative Teil der reellen Achse von $-\infty$ bis -1, wobei diese Achse zur oberen Halbebene zählt.

Für alle Funktionen aus Tabelle 5.6 ist im normalisierten Bereich der Betrag des relativen Fehlers kleiner als die jeweils angegebene Fehlerschranke. Fallen jedoch Real– oder Imaginärteil in den denormalisierten Bereich (3.2), d.h.

$$0 \leq |\mathrm{Re}(f(z))|, |\mathrm{Im}(f(z))| < 10^{-511} = \mathrm{MINLONGREAL},$$

und werden dadurch hintere, von Null verschiedene Ziffern der Ergebnismantisse gleich Null gesetzt oder entsteht Underflow mit dem relativen Fehler 1, so erfolgt die Fehlermeldung:

rel. error exceeds error bound

mit anschließendem Programmabbruch.

5.2.1.5 Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

procedure read (**var** f: text; **var** a: Lcomplex);
procedure write (**var** f: text; a: Lcomplex);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe einer komplexen Zahl $z = x + i \cdot y$ muß in der Form

$$(x, y) \qquad \text{oder in der Form} \qquad x$$

erfolgen. Im zweiten Fall wird der Imaginärteil y automatisch auf 0 gesetzt. x und y sind dabei *real*-Konstanten. Die Ausgabe einer komplexen Zahl erfolgt stets in der Form

$$(x, y)$$

mit dem Standardformat für die *Longreal*-Größen x und y aus Tabelle 3.4.

5.2.2 Exakte Auswertung komplexer Ausdrücke

In **PASCAL–XSC** können mit Hilfe der Operatoren $+, -, *$ Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer, real, complex, dotprecision* oder Produkte vom Typ *integer, real, complex* oder Skalarprodukte vom Typ *real* oder *complex* mit Hilfe des langen Akkus **rundungsfehlerfrei** berechnet und durch die Anweisungen

z := #*(exakter Ausdruck vom Typ *complex*);
z := #>(exakter Ausdruck vom Typ *complex*);
z := #<(exakter Ausdruck vom Typ *complex*);

mit nur einer einzigen Rundung bei Real- und Imaginärteil in eine Variable z vom Typ *complex* abgespeichert werden. Dieses #–Konzept steht auch für die BCD–Version in **vollem Umfang** zur Verfügung; es ist jedoch **nicht** anwendbar, wenn nur einer der Operanden im exakt auszuwertenden, komplexen Ausdruck vom Typ Longreal oder Lcomplex ist!

Um das #–Konzept für diese Datentypen wenigstens simulieren zu können, benötigt man den Datentyp **cdotprecision**:

type cdotprecision = **record** Re, Im: dotprecision **end**;

5.2.2.1 Operatoren

Wie auf Seite 42 bechrieben, muß auch jetzt z.B. für einen Additionsoperator mit Lcomplex–Operanden und einem Ergebnis vom Typ cdotprecision ein **neuer** Operatorname (plus_c) gewählt werden, da der normale + Operator mit den gleichen Operanden schon für den Ergebnistyp Lcomplex vergeben ist. In nachfolgender Tabelle 5.7 sind die entsprechenden neuen Operatornamen zusammengestellt:

Operation	Operator–Name	Ergebnistyp
Addition	**plus_c**	cdotprecision
Subtraktion	**minus_c**	cdotprecision
Multiplikation	**times_c**	cdotprecision

Tabelle 5.7: Operatornamen mit Ergebnistyp cdotprecision

rechter Operand	integer/real Longreal	complex Lcomplex	cdotprecision (dotprecision)
linker Operand			
integer/real Longreal	times_c plus_c, minus_c	times_c plus_c, minus_c	$*$ plus_c, minus_c
complex Lcomplex	times_c plus_c, minus_c	times_c plus_c, minus_c	$*$ plus_c, minus_c
cdotprecision (dotprecision)	$*$ plus_c, minus_c	$*$ plus_c, minus_c	$*$ plus_c, minus_c

Tabelle 5.8: Operatoren des Moduls LC_ARI mit Ergebnistyp *cdotprecision*

In Tabelle 5.8 sind für die Operatoren plus_c, minus_c, times_c, $*$ alle erlaubten Operanden–Typen angegeben. Mit einem *dotprecision*–Operanden existiert jedoch nur der $*$ Operator mit dem zweiten Operandentyp *cdotprecision*!

Wertzuweisungen an eine Variable vom Typ cdotprecision:
Durch Überladen des Zuweisungsoperators := sind folgende Wertzuweisungen möglich:

cdotprecision–Variable := Lcomplex–Ausdruck;
cdotprecision–Variable := complex–Ausdruck;
cdotprecision–Variable := dotprecision–Ausdruck;
cdotprecision–Variable := Longreal–Ausdruck;
cdotprecision–Variable := real/integer–Ausdruck;

5.2.2.2 Transferfunktionen

Zur Wandlung zwischen den Typen

integer/real/Longreal/dotprecision

einerseits und dem Typ *cdotprecision* andererseits werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
DOTCOMPL (x1,x2)	*cdotprecision*	$dc = x1 + i \cdot x2; \quad i = \sqrt{-1}$
DOTCOMPL (x)	*cdotprecision*	$dc = x + i \cdot 0; \quad i = \sqrt{-1}$
DOTCOMPL (d1,d2)	*cdotprecision*	$dc = d1 + i \cdot d2; \quad i = \sqrt{-1}$
DOTCOMPL (d)	*cdotprecision*	$dc = d + i \cdot 0; \quad i = \sqrt{-1}$
Re (dc)	*dotprecision*	Realteil von dc
Im (dc)	*dotprecision*	Imaginärteil von dc

x, x1, x2 = *integer/real/Longreal*–Ausdruck; dc = *cdotprecision*-Variable;
d, d1, d2 = *dotprecision*–Variable;

5.2.2.3 Simulation von SUM-Ausdrücken

Zur Auswertung von **SUM**–Ausdrücken im #–Konzept stehen die folgenden Datentypen und Prozeduren zur Verfügung:

- **type** rvector = **dynamic** array [*] of **real**;
- **type** Lrvector = **dynamic** array [*] of **Longreal**;
- **type** cvector = **dynamic** array [*] of **complex**;
- **type** Lcvector = **dynamic** array [*] of **Lcomplex**;

1. procedure **ADDNACCU** (var dc: cdotprecision; var a: rvector);
   Exakte Addition des real-Feldes a zur cdotprecision-Variablen dc;
2. procedure **ADDNACCU** (var dc: cdotprecision; var a: Lrvector);
   Exakte Addition des Longreal-Feldes a zur cdotprecision-Variablen dc;

3. procedure **ADDNACCU** (var dc: cdotprecision; var a: cvector);
   Exakte Addition des complex-Feldes a zur cdotprecision-Variablen dc;

4. procedure **ADDNACCU** (var dc: cdotprecision; var a: Lcvector);
   Exakte Addition des Lcomplex-Feldes a zur cdotprecision-Variablen dc;

5. procedure **SUBNACCU** (var dc: cdotprecision; var a: rvector);
   Exakte Subtraktion des real-Feldes a von der cdotprecision-Variablen dc;

6. procedure **SUBNACCU** (var dc: cdotprecision; var a: Lrvector);
   Exakte Subtraktion des Longreal-Feldes a von der cdotprecision-Variablen dc;

7. procedure **SUBNACCU** (var dc: cdotprecision; var a: cvector);
   Exakte Subtraktion des complex-Feldes a von der cdotprecision-Variablen dc;

8. procedure **SUBNACCU** (var dc: cdotprecision; var a: Lcvector);
   Exakte Subtraktion des Lcomplex-Feldes a von der cdotprecision-Variablen dc;

9. procedure **PADDNACCU** (var dc: cdotprecision; var a,b: rvector);
   Exakte Addition des real-Skalarproduktes zur cdotprecision-Variablen dc;

10. procedure **PADDNACCU** (var dc: cdotprecision; var a,b: Lrvector);
    Exakte Addition des Longreal-Skalarproduktes zur cdotprecision-Variablen dc;

11. procedure **PADDNACCU** (var dc: cdotprecision; var a,b: cvector);
    Exakte Addition des complex-Skalarproduktes zur cdotprecision-Variablen dc;

12. procedure **PADDNACCU** (var dc: cdotprecision; var a,b: Lcvector);
    Exakte Addition des Lcomplex-Skalarproduktes zur cdotprecision-Variablen dc;

13. procedure **PSUBNACCU** (var dc: cdotprecision; var a,b: rvector);
    Exakte Subtraktion des real-Skalarproduktes von der cdotprecision-Variablen dc;

14. procedure **PSUBNACCU** (var dc: cdotprecision; var a,b: Lrvector);
    Exakte Subtraktion des Longreal-Skalarproduktes von der cdotprecision-Variablen dc;

15. procedure **PSUBNACCU** (var dc: cdotprecision; var a,b: cvector);
    Exakte Subtraktion des complex-Skalarproduktes von der cdotprecision-Variablen dc;

16. procedure **PSUBNACCU** (var dc: cdotprecision; var a,b: Lcvector);
    Exakte Subtraktion des Lcomplex-Skalarproduktes von der cdotprecision-Variablen dc;

Mit Hilfe der folgenden Funktionen kann der Wert einer *cdotprecision*–Variablen durch die drei verschiedenen Rundungsarten in einen Wert vom Typ *Lcomplex* umgewandelt werden. Fallen dabei Real- oder Imaginärteil in den denormalisierten Zahlenbereich (3.2), so kann man mit den Funktionen

LONGNEXT_TEST, LONGUP_TEST, LONGDOWN_TEST

überprüfen, ob der aktuelle, relative Fehler bei den entsprechenden Rundungen die jeweiligen Fehlerschranken für den normalisierten Bereich übersteigen; vergleiche dazu die Bemerkungen von Seite 23.

- function **LONGNEXT** (var dc: cdotprecision): Lcomplex;
  y:=LONGNEXT(dc) liest den dc–Wert in die nächste Lcomplex–Zahl y;
- function **LONGDOWN** (var dc: cdotprecision): Lcomplex;
  y:=LONGDOWN(dc) liest den dc–Wert in die nächstkleinere Lcomplex–Zahl y;
- function **LONGUP** (var dc: cdotprecision): Lcomplex;
  y:=LONGUP(dc); liest den dc–Wert in die nächstgrößere Lcomplex–Zahl y;
- function **LONGNEXT_TEST** (var dc: cdotprecision;
  var error: boolean): Lcomplex;
  Liest den dc–Wert in die nächste Lcomplex–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $> 0.5 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).
- function **LONGUP_TEST** (var dc: cdotprecision;
  var error: boolean): Lcomplex;
  Liest den dc–Wert in die nächstgrößere Lcomplex–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).
- function **LONGDOWN_TEST** (var dc: cdotprecision;
  var error: boolean): Lcomplex;
  Liest den dc–Wert in die nächstkleinere Lcomplex–Zahl;
  error = TRUE $\Longleftrightarrow$ relativer Fehler $\geq 1 \cdot 10^{-20}$; (durch Nullsetzen hinterer Ziffern der Ergebnismantisse im Bereich (3.2)).

5.2.2.4 Beispiele

1. Beispiel:

```
program test (input,output);
var Lz      : Lcomplex;          a,b       : Lcvector[1..100];
    error   : boolean;           dc1,dc2   : cdotprecision;
    xl      : Longreal;          z         : complex;

begin
   ...
   dc1 := 0;     PADDNACCU(dc1,a,b);
   dc2 := xl times_c 3 minus_c Lz times_c z plus_c dc1;
   Lz := LONGNEXT_TEST (dc2,error);
end.
```

Obiges Programm berechnet den Ausdruck

$$dc2 = 3 \cdot xl - z \cdot Lz + \sum_{k=1}^{100} a[k] \cdot b[k]$$

rundungsfehlerfrei und schreibt seinen Wert mit **nur einer einzigen Rundung** in die nächstgelegene *Lcomplex*-Zahl der Variablen Lz. Falls beim Real- oder Imaginärteil durch das jeweilige Runden zur nächstgelegenen *Longreal*-Zahl der relative Betragsfehler größer wird als $\varepsilon(G) = 0.5 \cdot 10^{-20}$, erhält die Variable error den Wert TRUE.

2. Beispiel:

Mit Hilfe der Operatoren $*$ und **plus_c** zwischen *cdotprecision*- und *Lcomplex*-Operanden kann ein komplexwertiges Polynom

$$\mathrm{P}(z) = \sum_{k=0}^{n} a_k \cdot z^k$$

mit der folgenden Funktion nach dem Hornerschema **rundungsfehlerfrei** berechnet werden:

```
function HORNER_cdot (var a: Lcvector;
                          z: Lcomplex): cdotprecision;

var k : integer;
    y : cdotprecision;

begin
    y := a[ub(a)];
    for k := ub(a)-1 downto lb(a) do
    y := y * z plus_c a[k];
    HORNER := y
end;
```

Die Koeffizienten

$$\begin{array}{lll} a_0 = 120 + 90i; & a_1 = -54 + 22i; & a_2 = 6 + 7i; \\ a_3 = -5 - 3i; & a_4 = 1 + 0i; & \end{array}$$

realisieren ein komplexwertiges Polynom $\mathrm{P}(z)$ mit den Nullstellen:

$$z_{1,2} = \pm\sqrt{2} \cdot (1 + 2i), \quad z_3 = 5, \quad z_4 = 3i.$$

Mit den Deklarationen:

```
var a: Lcvector[0..4];
    z,y: Lcomplex;    dotz: cdotprecision;
```

und den Anweisungen:

```
z      := 5.000000000000000000001;    { z ≈ z3 }
dotz   := HORNER_cdot (a,z);
y      := LONGNEXT(dotz);
```

erhält man mit der BCD–Version das Ergebnis:

$$y = \widetilde{\mathrm{P}}(z) = 1.310000000000000000001 \cdot 10^{-18} - 1.3300\ldots00 \cdot 10^{-18} \cdot i$$

mit **nur einer Rundung** bzgl. Real- und Imaginärteil.

5.2.3 Der Typ rrGcomplex

Zur Implementierung **hochgenauer**, komplexwertiger Standardfunktionen mit dem 21-stelligen Ergebnistyp *Lcomplex* wurden die entsprechenden Algorithmen im **rrGcomplex**-Format realisiert, das wie folgt definiert ist:

```
type rrGcomplex = record   Re, Im : rrG   end;
```

Es wurden nur die wichtigsten Operatoren, Funktionen und Prozeduren bereitgestellt, die zur Realisierung der Standardfunktionen notwendig waren.

5.2.3.1 Operatoren

Neben den monadischen Operatoren $+ \; -$ existieren die in folgender Tabelle zusammengestellten, arithmetischen Grundoperatoren:

1. Operand		**2. Operand**	**Ergebnistyp**	**Rel. Fehler**
rrGcomplex	$+$	rrGcomplex	rrGcomplex	$5.0001 \cdot 10^{-26}$
rrGcomplex	$-$	rrGcomplex	rrGcomplex	$5.0001 \cdot 10^{-26}$
rrGcomplex	$*$	rrGcomplex	rrGcomplex	$1.1112 \cdot 10^{-25}$
real	$*$	rrGcomplex	rrGcomplex	$5.0001 \cdot 10^{-26}$
Longreal	$*$	rrGcomplex	rrGcomplex	$5.0001 \cdot 10^{-26}$
rrG	$*$	rrGcomplex	rrGcomplex	$5.0001 \cdot 10^{-26}$
rrGcomplex	$/$	rrGcomplex	rrGcomplex	$2.4617 \cdot 10^{-24}$
rrGcomplex	$/$	real	rrGcomplex	$2.2505 \cdot 10^{-24}$
rrGcomplex	$/$	Longreal	rrGcomplex	$2.2505 \cdot 10^{-24}$
rrGcomplex	$/$	rrG	rrGcomplex	$2.2505 \cdot 10^{-24}$

Als Vergleichsoperatoren existieren lediglich die Operatoren $= \; <>$.

5.2.3.2 Transferfunktionen

Zur Wandlung zwischen den Typen *rrG* und *rrGcomplex* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
RRGCOMPL (x1,x2)	*rrGcomplex*	$z = x1 + i \cdot x2; \quad i = \sqrt{-1}$
RRGCOMPL (x)	*rrGcomplex*	$z = x + i \cdot 0; \quad i = \sqrt{-1}$
RE (z)	*rrG*	z.Re = Realteil von z
IM (z)	*rrG*	z.Im = Imaginärteil von z

x, x1, x2 = *rrG*–Ausdruck; z = *rrGcomplex*–Ausdruck;

5.2.3.3 Standardfunktionen

Es stehen folgende Standardfunktionen mit Argumenttyp **rrGcomplex** und Ergebnistyp **rrGcomplex** zur Verfügung:

Aufruf	Ergebnistyp	Funktion	$\varepsilon(\mathrm{Re}\{f(z)\})$	$\varepsilon(\mathrm{Im}\{f(z)\})$
sqr (z)	rrGcomplex	$z^2 = z \cdot z$	$1.5001 \cdot 10^{-25}$	$1.5001 \cdot 10^{-25}$
sqrt (z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
sqrt_u (z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
sqrt_negim (z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
exp (z)	rrGcomplex	e^z	$1.5318 \cdot 10^{-24}$	$1.5318 \cdot 10^{-24}$
ln (z)	rrGcomplex	$\ln(z)$	$3.3317 \cdot 10^{-24}$	$5.3282 \cdot 10^{-24}$
ln1p (z)	rrGcomplex	$\ln(1+z)$	$3.8802 \cdot 10^{-24}$	$5.8285 \cdot 10^{-24}$
sin (z)	rrGcomplex	$\sin(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
cos (z)	rrGcomplex	$\cos(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
tan (z)	rrGcomplex	$\tan(z)$	$9.7041 \cdot 10^{-24}$	$1.1985 \cdot 10^{-23}$
cot (z)	rrGcomplex	$\cot(z)$	$9.7041 \cdot 10^{-24}$	$1.1985 \cdot 10^{-23}$
sinh (z)	rrGcomplex	$\sinh(z)$	$3.7017 \cdot 10^{-24}$	$2.7572 \cdot 10^{-24}$
cosh (z)	rrGcomplex	$\cosh(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
tanh (z)	rrGcomplex	$\tanh(z)$	$1.1985 \cdot 10^{-23}$	$9.7041 \cdot 10^{-24}$
coth (z)	rrGcomplex	$\coth(z)$	$1.1985 \cdot 10^{-23}$	$9.7041 \cdot 10^{-24}$
conj (z)	rrGcomplex	$\bar{z} = x - i \cdot y$	0.0	0.0
arg (z)	rrG	$\varphi\,; \quad z = r \cdot e^{i\cdot\varphi}\,;$	$5.3282 \cdot 10^{-24}$	
arg1p (z)	rrG	$\varphi\,; \quad 1+z = r \cdot e^{i\cdot\varphi}\,;$	$5.8285 \cdot 10^{-24}$	
abs (z)	rrG	$r = \sqrt{x^2+y^2}\,;$	$1.2754 \cdot 10^{-24}$	

Tabelle 5.9: Standardfunktionen des Moduls LC_ARI, z: rrGcomplex

Anmerkungen zu den Funktionen: sqrt(z), sqrt_u(z), sqrt_negim, ln(z), ln1p(z):

- **sqrt(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.
- **sqrt_u(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur unteren Halbebene zählt.
- **sqrt_negim(z)** Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse zur linken Halbebene zählt.
- **ln(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.

- **ln1p(z)** Der Verzweigungsschnitt ist der negative Teil der reellen Achse von $-\infty$ bis -1, wobei diese Achse zur oberen Halbebene zählt.

Es stehen folgende Standardfunktionen mit Argumenttyp **Lcomplex** und Ergebnistyp **rrGcomplex** zur Verfügung:

Aufruf	Ergebnistyp	Funktion	$\varepsilon(\mathrm{Re}\{f(z)\})$	$\varepsilon(\mathrm{Im}\{f(z)\})$
sqr_rrG (z)	rrGcomplex	$z^2 = z \cdot z$	$1.5001 \cdot 10^{-25}$	$1.5001 \cdot 10^{-25}$
sqrt_rrG (z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
sqrt_u_rrG (z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
sqrt_negim_rrG(z)	rrGcomplex	$\sqrt{z}$	$4.4551 \cdot 10^{-24}$	$4.4551 \cdot 10^{-24}$
exp_rrG (z)	rrGcomplex	e^z	$1.5318 \cdot 10^{-24}$	$1.5318 \cdot 10^{-24}$
ln_rrG (z)	rrGcomplex	$\ln(z)$	$5.2040 \cdot 10^{-24}$	$5.3282 \cdot 10^{-24}$
ln1p_rrG (z)	rrGcomplex	$\ln(1+z)$	$2.9430 \cdot 10^{-24}$	$5.8285 \cdot 10^{-24}$
sin_rrG (z)	rrGcomplex	$\sin(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
cos_rrG (z)	rrGcomplex	$\cos(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
tan_rrG (z)	rrGcomplex	$\tan(z)$	$9.7041 \cdot 10^{-24}$	$1.1985 \cdot 10^{-23}$
cot_rrG (z)	rrGcomplex	$\cot(z)$	$9.7041 \cdot 10^{-24}$	$1.1985 \cdot 10^{-23}$
sinh_rrG (z)	rrGcomplex	$\sinh(z)$	$3.7017 \cdot 10^{-24}$	$2.7572 \cdot 10^{-24}$
cosh_rrG (z)	rrGcomplex	$\cosh(z)$	$2.7572 \cdot 10^{-24}$	$3.7017 \cdot 10^{-24}$
tanh_rrG (z)	rrGcomplex	$\tanh(z)$	$1.1985 \cdot 10^{-23}$	$9.7041 \cdot 10^{-24}$
coth_rrG (z)	rrGcomplex	$\coth(z)$	$1.1985 \cdot 10^{-23}$	$9.7041 \cdot 10^{-24}$
conj_rrG (z)	rrGcomplex	$\bar{z} = x - iy$	0.0	0.0
arg_rrG (z)	rrG	φ ; $z = r \cdot e^{i\cdot\varphi}$;	$5.3282 \cdot 10^{-24}$	
arg1p_rrG (z)	rrG	φ ; $1 + z = r \cdot e^{i\cdot\varphi}$;	$5.8285 \cdot 10^{-24}$	
abs_rrG (z)	rrG	$r = \sqrt{x^2 + y^2}$;	$1.2504 \cdot 10^{-24}$	

Tabelle 5.10: Standardfunktionen des Moduls LC_ARI, z: Lcomplex

Anmerkungen zu den Funktionen: sqrt_rrG(z), sqrt_u_rrG(z), sqrt_negim_rrG, ln_rrG(z), ln1p_rrG(z):

- **sqrt_rrG(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.
- **sqrt_u_rrG(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur unteren Halbebene zählt.
- **sqrt_negim_rrG(z)** Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse zur linken Halbebene zählt.
- **ln_rrG(z)** Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse zur oberen Halbebene zählt.

- **ln1p_rrG(z)** Der Verzweigungsschnitt ist der negative Teil der reellen Achse von $-\infty$ bis -1, wobei diese Achse zur oberen Halbebene zählt.

5.2.3.4 Überladungen des Zuweisungsoperators

Die Wandlung zwischen den Typen *Lcomplex* und *rrGcomplex* wird in Form einer überladenen Zuweisung bereitgestellt:

Zuweisung	**Bedeutung**
Lc := rrGc	Rundung von rrGc zur nächsten Lcomplex–Zahl
rrGc := Lc	rrGc ≡ Lc, aber unterschiedliche Typen von rrGc, Lc

rrGc = *rrGcomplex*–Variable/Ausdruck; Lc = *Lcomplex*–Variable/Ausdruck;

5.2.3.5 Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

procedure read (**var** f: text; **var** z: rrGcomplex);
procedure write (**var** f: text; z: rrGcomplex);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe einer komplexen Zahl $z = x + i \cdot y$ muß in der Form

$$(x, y) \qquad \text{oder in der Form} \qquad x$$

erfolgen. Im zweiten Fall wird der Imaginärteil y automatisch auf 0 gesetzt. x und y sind dabei *rrG*-Werte, deren Eingabe auf Seite 56 beschrieben wird.

Die Ausgabe einer rrGcomplex–Zahl erfolgt stets in der Form

$$(x, y)$$

wobei z.B. der Realteil x : rrG stets in der Form

$$\{\, \text{x.r1,x.r2,x.G} \,\}$$

ausgegeben wird.

Kapitel 6

Komplexe Intervallarithmetik

Mit dem Modul **CI_ARI** werden die für das Rechnen mit komplexen Intervallen notwendigen Operatoren, Funktionen und Prozeduren bezüglich des Datentyps **REAL** bereitgestellt.

Das Modul **LCI_ARI** ermöglicht zusätzlich komplexe Intervallrechnungen auf Basis der Typen **REAL, LONGREAL, DOTPRECISION** und **RRG**.

6.1 Das Modul CI_ARI

Für Rechnungen mit komplexen Intervallen wird der Datentyp *cinterval* entsprechend der Definition

type cinterval = **record** re,im : interval **end**;

im Sprachkern von PASCAL–XSC bereitgestellt. Für z : *cinterval* sind damit Real- und Imaginärteil von z **Intervalle**, so daß

$$z = z.re + i \cdot z.im = [z.re.inf, z.re.sup] + i \cdot [z.im.inf, z.im.sup]; \qquad i = \sqrt{-1}$$

in der komplexen Ebene als Rechteck-Intervall gedeutet werden kann:

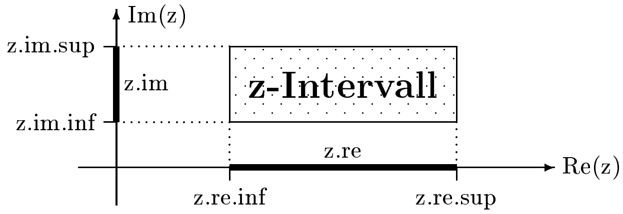


Abbildung 6.1: Komplexes Intervall

6.1.1 Operatoren

Sämtliche in diesem Modul vordefinierten arithmetischen und Verbands–Operatoren liefern den Ergebnistyp *cinterval*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der Rundung zum kleinsten, einschließenden komplexen Intervall vom Typ *cinterval* zur Verfügung.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind mengentheoretisch zu interpretieren. Dabei bedeutet:

= gleich	<> ungleich	< echte Teilmenge von
<= Teilmenge von	> echte Obermenge von	>= Obermenge von

Für v, w vom Typ *cinterval* gilt:

$$v \ <= \ w \iff (v.re \ <= \ w.re) \ \textbf{and} \ (v.im \ <= \ w.im).$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehung sind dabei die Operatoren für Intervalle vom Typ *interval*.

rechter Operand	integer real	complex	interval	cinterval
linker Operand				
monadisch				+, −
integer real		+*		◇ **in**, =, <> +*
complex	+*	+*	◇ **in**, =, <> +*	◇ **in**, =, <> +*
interval		◇ =, <> +*		◇ **in**, ∨, >< +*, **
cinterval	◇ =, <> +*	◇ =, <> +*	◇ ∨, >< +*, **	◇ **in**, ∨, >< +*, **

Tabelle 6.1: Die Operatoren des Moduls CI_ARI
$\diamond \in \{+, -, *, /\}$ $\quad \vee \in \{=, <>, <, <=, >, >=\}$

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in“ bzw. „enthalten im Innern“ sowie >< für den Test auf Disjunktheit zweier komplexer Intervalle zur Verfügung. Dabei heißen zwei komplexe Intervalle v, w disjunkt, wenn gilt: $v \cap w = \emptyset$ (leere Menge). Für zwei komplexe Intervalle x und y gilt:

$$x \textbf{ in } y \iff (x.re \textbf{ in } y.re) \textbf{ and } (x.im \textbf{ in } y.im).$$

Die Verbandsoperatoren $+*$ bzw. $**$ bezeichnen die Bildung der Intervall–Hülle bzw. des Durchschnitts, d.h. der Operator $+*$ liefert das kleinste, beide Operanden umfassende komplexe Intervall, und der Operator $**$ liefert das komplexe Schnittintervall. Ein leerer Schnitt führt zu einem Laufzeitfehler.

6.1.2 Transferfunktionen

Zur Wandlung zwischen den Typen *integer, real, complex, interval* und *cinterval* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	Bedeutung
compl (I1,I2)	*cinterval*	$\mathrm{I1} + i \cdot \mathrm{I2}$
compl (R,I)	*cinterval*	$\mathrm{R} + i \cdot \mathrm{I}$
compl (I,R)	*cinterval*	$\mathrm{I} + i \cdot \mathrm{R}$
compl (I)	*cinterval*	$\mathrm{I} + i \cdot 0$
intval(C1,C2)	*cinterval*	[C1.re,C2.re] $+ i \cdot$ [C1.im,C2.im];
		C1.re $\le$ C2.re **und** C1.im $\le$ C2.im
intval(R,C)	*cinterval*	[R,C.re] $+ i \cdot$ [0, C.im];
		R $\le$ C.re **und** 0 $\le$ C.im
intval(C,R)	*cinterval*	[C.re,R] $+ i \cdot$ [C.im, 0];
		C.re $\le$ R **und** C.im $\le$ 0
intval(C)	*cinterval*	[C.re,C.re] $+ i \cdot$ [C.im,C.im]
re (CI)	*interval*	Realteil von CI
im (CI)	*interval*	Imaginärteil von CI
inf (CI)	*complex*	Komplexe Untergrenze z von CI, mit:
		$z =$ (CI.re.inf, CI.im.inf)
sup (CI)	*complex*	Komplexe Obergrenze z von CI, mit:
		$z =$ (CI.re.sup, CI.im.sup)

Tabelle 6.2: Transferfunktionen des Moduls CI_ARI

R = *real*-Ausdruck, I, I1, I2 = *interval*-Ausdruck, $i = \sqrt{-1}$,
C, C1, C2 = *complex*-Ausdruck, CI = *cinterval*-Ausdruck

6.1.3 Überladungen des Zuweisungsoperators

Die Wandlungen von *integer/real/complex/interval* nach *cinterval* wird auch in Form überladener Zuweisungen bereitgestellt:

Zuweisung	**Bedeutung**
CI := R	CI := compl (intval(R))
CI := C	CI := intval (C)
CI := I	CI := compl (I)

Tabelle 6.3: Überladungen des Zuweisungsoperators

CI = *cinterval*-Variable, I = *interval*-Ausdruck,
C = *complex*-Ausdruck, R = *integer/real*-Ausdruck

6.1.4 Standardfunktionen

Es stehen die folgenden Standardfunktionen zur Verfügung:

Funktion	**Ergb.-Typ**	**Bedeutung**	
sqr (ci)	cinterval	$ci^2 = ci \cdot ci$	Quadrat
sqrt (ci)	cinterval	$\sqrt{ci}$	Quadratwurzel
exp (ci)	cinterval	e^{ci}	Exponentialfunktion
ln (ci)	cinterval	$\ln(ci)$	Natürlicher Logarithmus
ln1p (ci)	cinterval	$\ln(1 + ci)$	Natürlicher Logarithmus
power (w,z)	cinterval	w^z	Potenzfunktion, nicht immer hochgenau
xp1powy (w,z)	cinterval	$(1 + w)^z$	Potenzfunktion, nicht immer hochgenau
sin (ci)	cinterval	$\sin(ci)$	Sinus
cos (ci)	cinterval	$\cos(ci)$	Kosinus
sinh (ci)	cinterval	$\sinh(ci)$	Hyperbolischer Sinus
cosh (ci)	cinterval	$\cosh(ci)$	Hyperbolischer Kosinus
conj (ci)	cinterval	$\overline{ci} = x - i \cdot y$	Konjugation von $ci = x + i \cdot y$
arg (ci)	interval	φ	Argumentintervall für das ci-Rechteck

Funktion	Ergb.-Typ	Bedeutung	
arg1p (ci)	interval	φ	Argumentintervall für das $(1+ci)$-Rechteck
abs (ci)	interval	$\sqrt{ci.re^2 + ci.im^2}$	Absolutbetrag von ci
mid (ci)	complex	m	Mittelpunkt von ci
diam (ci)	real	d	Durchmesser von ci
blow (ci,r)	cinterval		Epsilonaufblähung

Tabelle 6.4: Standardfunktionen des Moduls CI_ARI

w,z,ci = *cinterval*-Ausdruck, r = *real*-Ausdruck, $i = \sqrt{-1}$

Anmerkungen zu den Standardfunktionen:

1. Mit Ausnahme von diam(ci), blow(ci, r), power(w, z), xp1powy(w, z) liefern alle anderen Funktionen aus Tabelle 6.4 **hochgenaue** Ergebnisse, d.h. nur in ganz seltenen Ausnahmefällen wird z.B. die untere Intervallgrenze von e^{ci} um 1 ulp zu klein berechnet.

2. x := diam (CI) liefert zum komplexen Intervall CI: *cinterval* eine Oberschranke x: *real* für die Länge L der entsprechenden Rechteckdiagonalen mit:

$$0 \leq L \leq x$$

   und einem relativen Höchstfehler: $\varepsilon(DIAM) = 1.0000004 \cdot 10^{-12}$;

3. function **sqrt** (var CI: cinterval): cinterval;
   CI sei ein komplexes Intervallargument mit:

$$CI = [x_1, x_2] + i \cdot [y_1, y_2], \qquad i = \sqrt{-1};$$

   Zu vorgegebenem CI liefert die Anweisung EI := sqrt(CI) ein komplexes Einschließungsintervall EI mit:

$$\sqrt{z} \in EI \qquad \text{für alle } z \in CI;$$

   Bezüglich der Lage des Verzweigungsschnitts in der komplexen z-Ebene werden drei Fälle unterschieden:

   (a) $y_1 < 0, \quad y_2 = 0$ und $x_1 < 0$;
   Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse der **unteren** Halbebene zugeordnet wird.

   (b) $y_1 < 0 < y_2$ und $x_1 < x_2 \leq 0$;
   Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse der **linken** Halbebene zugeordnet wird.

(c) In allen anderen Fällen ist der Verzweigungsschnitt wie üblich die negative, reelle Achse, wobei diese Achse der oberen Halbebene zugeordnet wird.

Durch die Wahl der Verzweigungsschnitte in den Fällen (a) und (b) wird eine unnötige Überschätzung der Einschließungsintervalle EI verhindert, die mit dem üblichen Verzweigungsschnitt im Fall (c) auftreten würde.

Allerdings geht in den Fällen (a) oder (b) die Inclusionsmonotonie verloren, denn es gilt z.B.:

$$A = \{[-2,-1] + i \cdot 0\} \subset \{[-2,-1] + i \cdot [-1,0]\} = B, \quad \text{aber} \quad \mathrm{sqrt}(A) \not\subset \mathrm{sqrt}(B).$$

4. function **ARG** (var CI: cinterval): interval;
CI sei ein komplexes Intervallargument mit:

$$CI = [x_1, x_2] + i \cdot [y_1, y_2], \qquad i = \sqrt{-1};$$

Zu vorgegebenem CI liefert die Anweisung $EI := \mathrm{ARG}(CI)$ ein Einschließungsintervall EI mit:

$$\arg(z) \in EI \qquad \text{für alle } z \in CI;$$

Gilt $CI = 0$, oder ist 0 **Innenpunkt** von CI, so erhält man das Intervall $EI = [-\pi, +\pi]$, wobei die Intervallgrenzen entsprechend gerundet werden.

In den restlichen Fällen müssen noch drei prinzipielle Lagen des komplexen Argumentintervalls unterschieden werden, in denen jeweils ein geeigneter Verzweigungsschnitt zu wählen ist:

(a) $y_1 < 0$, $y_2 = 0$ und $x_1 < 0$;
Der Verzweigungsschnitt ist die negative, reelle Achse, wobei diese Achse der **unteren** Halbebene zugeordnet wird.

(b) $y_1 < 0 < y_2$ und $x_1 < x_2 \leq 0$;
Der Verzweigungsschnitt ist die negative, imaginäre Achse, wobei diese Achse der **linken** Halbebene zugeordnet wird.

(c) In allen anderen Fällen ist der Verzweigungsschnitt wie üblich die negative, reelle Achse, wobei diese Achse der oberen Halbebene zugeordnet wird.

Durch die Wahl der Verzweigungsschnitte in den Fällen (a) und (b) wird eine unnötige Überschätzung der Einschließungsintervalle EI verhindert, die mit dem üblichen Verzweigungsschnitt im Fall (c) auftreten würde.

Allerdings geht in den Fällen (a) oder (b) die Inclusionsmonotonie verloren, denn es gilt z.B.:

$$A = \{[-2,-1] + i \cdot 0\} \subset \{[-2,-1] + i \cdot [-1,0]\} = B, \quad \text{aber} \quad \mathrm{ARG}(A) \not\subset \mathrm{ARG}(B).$$

5. function **ARG1P** (var CI: cinterval): interval;
CI sei ein komplexes Intervallargument mit:

$$CI = [x_1, x_2] + i \cdot [y_1, y_2], \qquad i = \sqrt{-1};$$

Zu vorgegebenem CI liefert die Anweisung EI := ARG1P(CI) ein Einschließungsintervall EI mit:

$$\arg(1+z) \in EI \qquad \text{für alle } z \in CI;$$

Gilt $CI = -1$, oder ist -1 **Innenpunkt** von CI, so erhält man das Intervall $EI = [-\pi, +\pi]$, wobei die Intervallgrenzen entsprechend gerundet werden.

In den restlichen Fällen müssen noch drei prinzipielle Lagen des komplexen Argumentintervalls unterschieden werden, in denen jeweils ein geeigneter Verzweigungsschnitt zu wählen ist:

(a) $y_1 < 0, \quad y_2 = 0$ und $x_1 < -1$;
Der Verzweigungsschnitt ist das Intervall $(-\infty, -1]$ auf der negativen, reellen Achse, wobei dieses Intervall der **unteren** Halbebene zugeordnet wird.

(b) $y_1 < 0 < y_2$ und $x_1 < x_2 \leq 0$;
Der Verzweigungsschnitt ist die Parallele zur negativen, imaginären Achse durch $z = -1 + 0 \cdot i$, wobei diese Parallele der **linken** Halbebene zugeordnet wird.

(c) In allen anderen Fällen ist der Verzweigungsschnitt wieder das Intervall $(-\infty, -1]$ auf der negativen, reellen Achse, wobei dieses Intervall der oberen Halbebene zugeordnet wird.

Durch die Wahl der Verzweigungsschnitte in den Fällen (a) und (b) wird eine unnötige Überschätzung der Einschließungsintervalle EI verhindert, die mit dem üblichen Verzweigungsschnitt im Fall (c) auftreten würde.

Allerdings geht in den Fällen (a) oder (b) die Inclusionsmonotonie verloren, denn es gilt z.B.:

$$A = \{[-3, -2] + i \cdot 0\} \subset \{[-3, -2] + i \cdot [-1, 0]\} = B, \qquad \text{aber}$$

$$\text{ARG1P}(A) \not\subset \text{ARG1P}(B).$$

6. function **LN** (var CI: cinterval): cinterval;
Zu vorgegebenem Argument CI liefert die Anweisung EI := LN(CI) ein Einschließungsintervall EI mit:

$$\ln(z) \in EI \qquad \text{für alle } z \in CI;$$

Enthält das komplexe Argumentintervall CI den Ursprung, so erfolgt eine Fehlermeldung mit Programmabbruch. Wegen $\text{Im}\{\text{LN}(CI)\} \equiv \text{ARG}(CI)$ gelten für die Funktionen LN(CI) und ARG(CI) bzgl. der Verzweigungsschnitte **identische** Aussagen!

7. function **LN1P** (var CI: cinterval): cinterval;
   Zu vorgegebenem CI liefert die Anweisung $EI := \text{LN1P}(CI)$ ein Einschließungsintervall EI mit:

   $$\ln(1+z) \in EI \qquad \text{für alle } z \in CI;$$

   Enthält das komplexe Argumentintervall CI die Zahl -1, so erfolgt eine Fehlermeldung mit Programmabbruch.
   Wegen $\text{Im}\{\text{LN1P}(CI)\} \equiv \text{ARG1P}(CI)$ gelten für die Funktionen $\text{LN1P}(CI)$ und $\text{ARG1P}(CI)$ bezüglich der Verzweigungsschnitte **identische** Aussagen!

8. function **POWER** (var w,z: cinterval): cinterval;
   Zu vorgegebenen komplexen Intervallargumenten w, z liefert die Anweisung $EI := \text{POWER}(w, z)$ ein nicht immer hochgenau berechnetes Einschließungsintervall EI mit:

   $$w^z \in EI \qquad \text{für alle komplexen Zahlen } u \in w \text{ und } v \in z;$$

   Enthält das komplexe Argumentintervall w die Zahl 0, so erfolgt eine Fehlermeldung mit Programmabbruch. Wegen

   $$w^z \equiv e^{z \cdot \ln(w)}$$

   gelten für die Funktionen $\text{POWER}(w, z), \text{LN}(w), \text{ARG}(w)$ bezüglich der Verzweigungsschnitte **identische** Aussagen!

9. function **XP1POWY** (var w,z: cinterval): cinterval;
   Zu vorgegebenen komplexen Intervallargumenten w, z liefert die Anweisung $EI := \text{XP1POWY}(w, z)$ ein nicht immer hochgenau berechnetes Einschließungsintervall EI mit:

   $$(1+w)^z \in EI \qquad \text{für alle komplexen Zahlen } u \in w \text{ und } v \in z;$$

   Enthält das komplexe Argumentintervall w die Zahl -1, so erfolgt eine Fehlermeldung mit Programmabbruch. Wegen

   $$(1+w)^z \equiv e^{z \cdot \ln(1+w)}$$

   gelten für die Funktionen $\text{XP1POWY}(w, z), \text{LN1P}(w), \text{ARG1P}(w)$ bezüglich der Verzweigungsschnitte **identische** Aussagen!

6.1.5 Ein-/Ausgabeanweisungen

Es stehen die bekannten Prozeduren

> **procedure** read (**var** f: text; **var** ci: cinterval);
> **procedure** write (**var** f: text; ci: cinterval);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Intervalls $ci = [x, y] + i \cdot [v, w]$ kann mit dem Prozeduraufruf **read**(ci) in der Form

$([x, y], [v, w])$ allgemeines komplexes Intervall

oder in der Form

$(x, [v, w])$ mit Punktintervall als Realteil, d.h. $x = y$

oder in der Form

$([x, y], v)$ mit Punktintervall als Imaginärteil, d.h. $v = w$

oder in der Form

$[x, y]$ rein reelles Intervall, d.h. $v = w = 0$

oder in der Form

(x, v) komplexes Punktintervall, d.h. $x = y$ und $v = w$

oder in der Form

x rein reelles Punktintervall, d.h. $x = y$ und $v = w = 0$

erfolgen.

Die Ausgabe eines komplexen Intervalls ci erfolgt durch den Prozeduraufruf **write**(ci) stets in der Form

$$([x, y], [v, w])$$

Das Ausgabeformat für die Real- und Imaginärteilintervalle ist dabei bestimmt durch das entsprechende Ausgabeformat für Intervalle, verleiche Seite 80.

Die Ausgabe der Intervallgrenzen x, y und v, w kann natürlich auch erfolgen durch:

write(ci.re.inf,ci.re.sup) und write(ci.im.inf,ci.im.sup);

wobei entsprechende Formatspezifikationen für die *real*-Zahlen

$$x = ci.re.inf, \quad y = ci.re.sup, \qquad v = ci.im.inf, \quad w = ci.im.sup$$

wieder möglich sind.

6.2 Das Modul LCI_ARI

6.2.1 Der Typ Lcinterval

Für allgemeine Gleitkommarechnungen mit komplexen Intervallen wird neben dem Typ *cinterval* zusätzlich der Datentyp *Lcinterval* entsprechend der Definition

type Lcinterval = **record** re,im : Linterval **end**;

bereitgestellt. Für z : *Lcinterval* sind damit Real- und Imaginärteil von z **Intervalle**, so daß

$$z = z.re + i \cdot z.im = [z.re.inf, z.re.sup] + i \cdot [z.im.inf, z.im.sup]; \qquad i = \sqrt{-1}$$

in der komplexen Ebene als Rechteck-Intervall gedeutet werden kann:

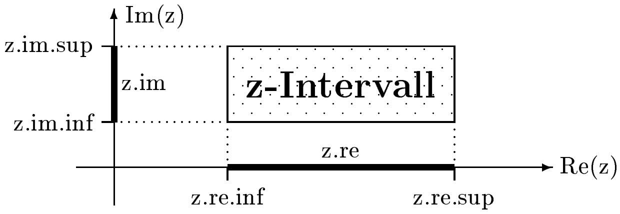


Abbildung 6.2: Komplexes Intervall

6.2.1.1 Operatoren

Wir betrachten zunächst alle in diesem Modul vordefinierten arithmetischen und Verbands–Operatoren mit dem Ergebnistyp *Lcinterval*. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der Rundung zum kleinsten, einschließenden komplexen Intervall vom Typ *Lcinterval* zur Verfügung.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind mengentheoretisch zu interpretieren. Dabei bedeutet:

= gleich	<> ungleich	< echte Teilmenge von
<= Teilmenge von	> echte Obermenge von	>= Obermenge von

Für v, w vom Typ *Lcinterval* gilt:

$$v \ <= \ w \iff (v.re \ <= \ w.re) \ \textbf{and} \ (v.im \ <= \ w.im).$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehung sind dabei die Operatoren für Intervalle vom Typ *Linterval*.

rechter Operand	integer real	Longreal	complex	Lcomplex
linker Operand	Ergebnistypen: *Lcinterval* oder *boolean*			
monadisch				
integer real				$+*$
Longreal			$+*$	$+*$
complex		$+*$		$+*$
Lcomplex	$+*$	$+*$	$+*$	$+*$
interval				$\diamond$ $=, <>$ $+*$
Linterval			$\diamond$ $=, <>$ $+*$	$\diamond$ $=, <>$ $+*$
cinterval		$\diamond$ $=, <>$ $+*$		$\diamond$ $=, <>$ $+*$
Lcinterval	$\diamond$ $=, <>$ $+*$	$\diamond$ $=, <>$ $+*$	$\diamond$ $=, <>$ $+*$	$\diamond$ $=, <>$ $+*$

Tabelle 6.5: Die Operatoren des Moduls LCI_ARI (Teil 1)

$$\diamond \in \{+,-,*,/\} \qquad \vee \in \{=,<>,<,<=,>,>=\}$$

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in" bzw. „enthalten im Innern" sowie $><$ für den Test auf Disjunktheit zweier komplexer Intervalle zur Verfügung. Dabei heißen zwei komplexe Intervalle v, w disjunkt, wenn gilt: $v \cap w = \emptyset$ (leere Menge). Für zwei komplexe Intervalle x und y gilt:

$$x \ \mathbf{in} \ y \iff (x.re \ \mathbf{in} \ y.re) \ \mathbf{and} \ (x.im \ \mathbf{in} \ y.im).$$

Die Verbandsoperatoren $+*$ bzw. $**$ bezeichnen die Bildung der Intervall–Hülle bzw. des Durchschnitts, d.h. der Operator $+*$ liefert das kleinste, beide Operanden umfassende komplexe Intervall, und der Operator $**$ liefert das komplexe Schnittintervall. Ein leerer Schnitt führt zu einem Laufzeitfehler.

rechter Operand	interval	Linterval	cinterval	Lcinterval
linker Operand	Bezeichnungen wie in Tabelle 6.5			
monadisch				$+, -$
integer real				$\diamond$ **in**, $=$, $<>$ $+*$
Longreal			$\diamond$ **in**, $=$, $<>$ $+*$	$\diamond$ **in**, $=$, $<>$ $+*$
complex		$\diamond$ **in**, $=$, $<>$ $+*$		$\diamond$ **in**, $=$, $<>$ $+*$
Lcomplex	$\diamond$ **in**, $=$, $<>$ $+*$	$\diamond$ **in**, $=$, $<>$ $+*$	$\diamond$ **in**, $=$, $<>$ $+*$	$\diamond$ **in**, $=$, $<>$ $+*$
interval				$\diamond$ **in**, $\vee$, $><$ $+*$, $**$
Linterval			$\diamond$ **in**, $\vee$, $><$ $+*$, $**$	$\diamond$ **in**, $\vee$, $><$ $+*$, $**$
cinterval		$\diamond$ $\vee$, $><$ $+*$, $**$		$\diamond$ **in**, $\vee$, $><$ $+*$, $**$
Lcinterval	$\diamond$ $\vee$, $><$ $+*$, $**$	$\diamond$ $\vee$, $><$ $+*$, $**$	$\diamond$ **in**, $\vee$, $><$ $+*$, $**$	$\diamond$ **in**, $\vee$, $><$ $+*$, $**$

Tabelle 6.6: Die Operatoren des Moduls LCI_ARI (Teil 2)

6.2.1.2 Transferfunktionen

Zur Wandlung zwischen den Typen *integer*, *real*, *Longreal*, *complex*, *Lcomplex*, *interval*, *Linterval*, *cinterval* und *Lcinterval* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
LCOMPL (I1,I2)	*Lcinterval*	$I1 + i \cdot I2$
LCOMPL (R,I)	*Lcinterval*	$R + i \cdot I$
LCOMPL (I,R)	*Lcinterval*	$I + i \cdot R$
LCOMPL (I)	*Lcinterval*	$I + i \cdot 0$
LCOMPL (I,LI)	*Lcinterval*	$I + i \cdot LI$
LCOMPL (LI,I)	*Lcinterval*	$LI + i \cdot I$
LCOMPL (LI1,LI2)	*Lcinterval*	$LI1 + i \cdot LI2$
LCOMPL (R,LI)	*Lcinterval*	$R + i \cdot LI$
LCOMPL (LI,R)	*Lcinterval*	$LI + i \cdot R$
LCOMPL (LR,I)	*Lcinterval*	$LR + i \cdot I$
LCOMPL (I,LR)	*Lcinterval*	$I + i \cdot LR$
LCOMPL (LR,LI)	*Lcinterval*	$LR + i \cdot LI$
LCOMPL (LI,LR)	*Lcinterval*	$LI + i \cdot LR$
LCOMPL (LI)	*Lcinterval*	$LI + i \cdot 0$
LINTVAL (C1,C2)	*Lcinterval*	$[C1.re, C2.re] + i \cdot [C1.im, C2.im]$;
		$C1.re \le C2.re$ **und** $C1.im \le C2.im$
LINTVAL (C,LC)	*Lcinterval*	$[C.re, LC.re] + i \cdot [C.im, LC.im]$;
		$C.re \le LC.re$ **und** $C.im \le LC.im$
LINTVAL (LC,C)	*Lcinterval*	$[LC.re, C.re] + i \cdot [LC.im, C.im]$;
		$LC.re \le C.re$ **und** $LC.im \le C.im$
LINTVAL (LC1,LC2)	*Lcinterval*	$[LC1.re, LC2.re] + i \cdot [LC1.im, LC2.im]$;
		$LC1.re \le LC2.re$ **und** $LC1.im \le LC2.im$
LINTVAL (R,C)	*Lcinterval*	$[R, C.re] + i \cdot [0, C.im]$;
		$R \le C.re$ **und** $0 \le C.im$

Tabelle 6.7: Transferfunktionen des Moduls LCI_ARI (Teil 1)

Funktion	**Ergebnistyp**	**Bedeutung**
LINTVAL (R,LC)	*Lcinterval*	$[R, LC.re] + i \cdot [0, LC.im]$;
		$R \leq LC.re$ **und** $0 \leq LC.im$
LINTVAL (LR,C)	*Lcinterval*	$[LR, C.re] + i \cdot [0, C.im]$;
		$LR \leq C.re$ **und** $0 \leq C.im$
LINTVAL (LR,LC)	*Lcinterval*	$[LR, LC.re] + i \cdot [0, LC.im]$;
		$LR \leq LC.re$ **und** $0 \leq LC.im$
LINTVAL (C,R)	*Lcinterval*	$[C.re, R] + i \cdot [C.im, 0]$;
		$C.re \leq R$ **und** $C.im \leq 0$
LINTVAL (LC,R)	*Lcinterval*	$[LC.re, R] + i \cdot [LC.im, 0]$;
		$LC.re \leq R$ **und** $LC.im \leq 0$
LINTVAL (C,LR)	*Lcinterval*	$[C.re, LR] + i \cdot [C.im, 0]$;
		$C.re \leq LR$ **und** $C.im \leq 0$
LINTVAL (LC,LR)	*Lcinterval*	$[LC.re, LR] + i \cdot [LC.im, 0]$;
		$LC.re \leq LR$ **und** $LC.im \leq 0$
LINTVAL (C)	*Lcinterval*	$[C.re, C.re] + i \cdot [C.im, C.im]$
LINTVAL (LC)	*Lcinterval*	$[LC.re, LC.re] + i \cdot [LC.im, LC.im]$
LINTVAL (CI)	*Lcinterval*	LONG$(CI.re) + i \cdot$ LONG$(CI.im)$
LONG (CI)	*Lcinterval*	LONG$(CI.re) + i \cdot$ LONG$(CI.im)$
SHORT (LCI)	*cinterval*	SHORT$(LCI.re) + i \cdot$ SHORT$(LCI.im)$
RE (LCI)	*Linterval*	Realteil von LCI
IM (LCI)	*Linterval*	Imaginärteil von LCI
INF (LCI)	*Lcomplex*	Komplexe Untergrenze z von LCI, mit:
		$z = (LCI.re.inf,\ LCI.im.inf)$
SUP (LCI)	*Lcomplex*	Komplexe Obergrenze z von LCI, mit:
		$z = (LCI.re.sup,\ LCI.im.sup)$

Tabelle 6.8: Transferfunktionen des Moduls LCI_ARI (Teil 2)

LC,LC1,LC2 = *Lcomplex*-Ausdruck, LCI = *Lcinterval*-Ausdruck,
C,C1,C2 = *complex*-Ausdruck, CI = *cinterval*-Ausdruck, $i = \sqrt{-1}$,
I,I1,I2 = *interval*-Ausdruck, R = *real,Longreal*-Ausdruck,
LI,LI1,LI2 = *Linterval*-Ausdruck, LR = *Longreal*-Ausdruck

6.2.1.3 Überladungen des Zuweisungsoperators

Die Wandlungen der Typen *integer*, *real*, *Longreal*, *complex*, *Lcomplex*, *interval*, *Linterval* nach *Lcinterval* wird auch in Form überladener Zuweisungen bereitgestellt:

Zuweisung	**Bedeutung**
LCI := R	LCI := LCOMPL (INTVAL(R))
LCI := LR	LCI := LCOMPL (INTVAL(LR))
LCI := CI	LCI := LINTVAL (CI)
LCI := C	LCI := LINTVAL (C)
LCI := LC	LCI := LINTVAL (LC)
LCI := I	LCI := LCOMPL (I)
LCI := LI	LCI := LCOMPL (LI)

Tabelle 6.9: Überladungen des Zuweisungsoperators

LCI = *Lcinterval*-Variable, R = *integer/real*-Ausdruck,
LR = *Longreal*-Ausdruck, CI = *cinterval*-Ausdruck,
C = *complex*-Ausdruck, LC = *Lcomplex*-Ausdruck,
I = *interval*-Ausdruck, LI = *Linterval*-Ausdruck;

6.2.1.4 Standardfunktionen

Es stehen die folgenden Standardfunktionen zur Verfügung:

Funktion	**Ergb.-Typ**	**Bedeutung**	
sqr (LCI)	Lcinterval	$LCI^2 = LCI \cdot LCI$	Quadrat
sqrt (LCI)	Lcinterval	$\sqrt{LCI}$	Quadratwurzel
exp (LCI)	Lcinterval	e^{LCI}	Exponentialfunktion
ln (LCI)	Lcinterval	$\ln(LCI)$	Natürlicher Logarithmus
ln1p (LCI)	Lcinterval	$\ln(1 + LCI)$	Natürlicher Logarithmus
power (W,Z)	Lcinterval	W^Z	Potenzfunktion, nicht immer hochgenau
xp1powy (W,Z)	Lcinterval	$(1 + W)^Z$	Potenzfunktion, nicht immer hochgenau

Funktion	**Ergb.-Typ**	**Bedeutung**	
sin (LCI)	Lcinterval	$\sin(LCI)$	Sinus
cos (LCI)	Lcinterval	$\cos(LCI)$	Kosinus
sinh (LCI)	Lcinterval	$\sinh(LCI)$	Hyperbolischer Sinus
cosh (LCI)	Lcinterval	$\cosh(LCI)$	Hyperbolischer Kosinus
conj (LCI)	Lcinterval	$\overline{LCI} = x - i \cdot y$	Konjugation von $LCI = x + i \cdot y$
arg (LCI)	Linterval	φ	Argumentintervall für das LCI-Rechteck
arg1p (LCI)	Linterval	φ	Argumentintervall für das $(1 + LCI)$-Rechteck
abs (LCI)	Linterval	$\sqrt{LCI.re^2 + LCI.im^2}$	Absolutbetrag von LCI
mid (LCI)	Lcomplex	m	Mittelpunkt von LCI
diam (LCI)	Longreal	d	Durchmesser von LCI
blow (LCI,R)	Lcinterval		Epsilonaufblähung

Tabelle 6.10: Standardfunktionen des Moduls LCI_ARI

W,Z,LCI = *Lcinterval*-Ausdruck, R = *real*-Ausdruck, $i = \sqrt{-1}$

Anmerkungen zu den Standardfunktionen:

1. Mit Ausnahme der Funktionen power, xp1powy sind alle Intervallfunktionen aus obiger Tabelle 6.10 **hochgenau**.

2. Für die Funktionen **sqrt, arg, arg1p, ln, ln1p, power, xp1powy** vom Ergebnistyp *Lcinterval* gelten bezüglich der Verzweigungsschnitte in Abhängigkeit von der Lage des komplexen Argumentintervalls die gleichen Aussagen wie für die entsprechenden *cinterval*-Funktionen von Seite 131.

3. xl := diam (LCI) liefert zum komplexen Intervall LCI: *Lcinterval* eine Oberschranke xl: *Longreal* für die Länge L der entsprechenden Rechteckdiagonalen mit: $0 \leq L \leq xl$;

4. Die Anweisung mc := MID(LCI) liefert den maximalgenauen komplexen Mittelpunkt mc: *Lcomplex* des komplexen Intervalls LCI: *Lcinterval*.

5. Bei Einschließungsmethoden wird oft die Funktion **blow** für die sogenannte Epsilonaufblähung benötigt. Mit einem *real*-Ausdruck R wirkt die blow(LCI,R)-Funktion auf die Intervalle LCI.re und LCI.im wie die entsprechende blow-Funktion aus dem Modul LI_ARI, vergl. Seite 84.

6.2.1.5 Ein-/Ausgabeanweisungen

Es stehen die bekannten Prozeduren

> **procedure** read (**var** f: text; **var** LCI: Lcinterval);
> **procedure** write (**var** f: text; LCI: Lcinterval);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Intervalls $LCI = [x, y] + i \cdot [v, w]$ kann mit dem Prozeduraufruf **read**(LCI) in der Form

$([x, y], [v, w])$	allgemeines komplexes Intervall

oder in der Form

$(x, [v, w])$	mit Punktintervall als Realteil, d.h. $x = y$

oder in der Form

$([x, y], v)$	mit Punktintervall als Imaginärteil, d.h. $v = w$

oder in der Form

$[x, y]$	rein reelles Intervall, d.h. $v = w = 0$

oder in der Form

(x, v)	komplexes Punktintervall, d.h. $x = y$ und $v = w$

oder in der Form

x	rein reelles Punktintervall, d.h. $x = y$ und $v = w = 0$

erfolgen.

Die Ausgabe eines komplexen Intervalls LCI erfolgt durch den Prozeduraufruf **write**(LCI) stets in der Form

$$([x, y], [v, w])$$

Das Ausgabeformat für die Real- und Imaginärteilintervalle ist dabei bestimmt durch das entsprechende Ausgabeformat für Intervalle, verleiche Seite 80.

Die Ausgabe der Intervallgrenzen x, y und v, w kann natürlich auch erfolgen durch:

write(LCI.re.inf,LCI.re.sup) und write(LCI.im.inf,LCI.im.sup);

wobei entsprechende Formatspezifikationen für die *Longreal*-Zahlen

$$x = LCI.re.inf, \;\; y = LCI.re.sup, \qquad v = LCI.im.inf, \;\; w = LCI.im.sup$$

wieder möglich sind.

6.2.2 Exakte komplexe Intervallausdrücke

In **PASCAL–XSC** können mit Hilfe der Operatoren $+, -, *$ komplexe Intervall–Ausdrücke vom Typ *cinterval* mit Hilfe des langen Akkumulators rundungsfehlerfrei berechnet und durch die Anweisung

CI := ##(exakter, komplexer Intervallausdruck);

mit nur einer einzigen Rundung pro Intervallgrenze in eine Variable CI vom Typ *cinterval* abgespeichert werden. Dieses ##–Konzept steht auch für die BCD–Version in **vollem Umfang** zur Verfügung; es ist jedoch **nicht** anwendbar, wenn nur einer der Operanden im exakt auszuwertenden, komplexen Intervallausdruck vom Typ *Longreal, Lcomplex, cdotprecision, Linterval, idotprecision, Lcinterval* oder *cidotprecision* ist!

Um das ##–Konzept für diese Datentypen wenigstens simulieren zu können, benötigt man den Datentyp **cidotprecision**:

type cidotprecision = **record** re, im: idotprecision **end**;

6.2.2.1 Transferfunktionen

Zur Wandlung zwischen den Typen

integer, real, Longreal, dotprecision, interval, Linterval, idotprecision, complex, Lcomplex, cdotprecision, cinterval, Lcinterval

einerseits und dem Typ *cidotprecision* andererseits werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	Bedeutung
DOTICOMPL(I1,I2)	*cidotprecision*	$I1 + i \cdot I2$;
DOTICOMPL(LI1,LI2)	*cidotprecision*	$LI1 + i \cdot LI2$;
DOTICOMPL(DI1,DI2)	*cidotprecision*	$DI1 + i \cdot DI2$;
DOTICOMPL(R,I)	*cidotprecision*	$[R, R] + i \cdot I$;
DOTICOMPL(L,LI)	*cidotprecision*	$[L, L] + i \cdot LI$;
DOTICOMPL(D,DI)	*cidotprecision*	$[D, D] + i \cdot DI$;
DOTICOMPL(I,R)	*cidotprecision*	$I + i \cdot [R, R]$;
DOTICOMPL(LI,L)	*cidotprecision*	$LI + i \cdot [L, L]$;
DOTICOMPL(DI,D)	*cidotprecision*	$DI + i \cdot [D, D]$;
DOTICOMPL(I)	*cidotprecision*	$I + i \cdot [0, 0]$;

Tabelle 6.11: Transferfunktionen mit dem Typ cidotprecision (Teil 1)

Funktion	Ergebnistyp	Bedeutung
DOTICOMPL(LI)	*cidotprecision*	$LI + i \cdot [0,0]$;
DOTICOMPL(DI)	*cidotprecision*	$DI + i \cdot [0,0]$;
DOTICOMPL(DC1,DC2)	*cidotprecision*	$[DC1.re, DC2.re]+$ $+i \cdot [DC1.im, DC2.im]$;
DOTICOMPL(LC1,LC2)	*cidotprecision*	$[LC1.re, LC2.re]+$ $+i \cdot [LC1.im, LC2.im]$;
DOTICOMPL(C1,C2)	*cidotprecision*	$[C1.re, C2.re]+$ $+i \cdot [C1.im, C2.im]$;
DOTICOMPL(D,DC)	*cidotprecision*	$[D, DC.re] + i \cdot [0, DC.im]$;
DOTICOMPL(L,LC)	*cidotprecision*	$[L, LC.re] + i \cdot [0, LC.im]$;
DOTICOMPL(R,C)	*cidotprecision*	$[R, C.re] + i \cdot [0, C.im]$;
DOTICOMPL(DC,D)	*cidotprecision*	$[DC.re, D] + i \cdot [DC.im, 0]$;
DOTICOMPL(LC,L)	*cidotprecision*	$[LC.re, L] + i \cdot [LC.im, 0]$;
DOTICOMPL(C,R)	*cidotprecision*	$[C.re, R] + i \cdot [C.im, 0]$;
DOTICOMPL(DC)	*cidotprecision*	$[DC.re, DC.re]+$ $+i \cdot [DC.im, DC.im]$;
DOTICOMPL(LC)	*cidotprecision*	$[LC.re, LC.re]+$ $+i \cdot [LC.im, LC.im]$;
DOTICOMPL(C)	*cidotprecision*	$[C.re, C.re] + i \cdot [C.im, C.im]$;
RE(DCI)	*idotprecision*	$[DCI.re.inf, DCI.re.sup]$;
IM(DCI)	*idotprecision*	$[DCI.im.inf, DCI.im.sup]$;
INF(DCI)	*cdotprecision*	$DCI.re.inf + i \cdot DCI.im.inf$;
SUP(DCI)	*cdotprecision*	$DCI.re.sup + i \cdot DCI.im.sup$;
LONG(DCI)	*Lcinterval*	Long$(DCI.re)+$ $+i \cdot$ Long$(DCI.im)$
SHORT(DCI)	*cinterval*	SHORT$(DCI.re)+$ $+i \cdot$ SHORT$(DCI.im)$

Tabelle 6.12: Transferfunktionen mit dem Typ cidotprecision (Teil 2)

I,I1,I2 = *interval*-Ausdruck, LI,LI1,LI2 = *Linterval*-Ausdruck,
DI,DI1,DI2 = *idotprecision*-Variable, R = *integer/real*-Ausdruck,
L = *Longreal*-Ausdruck, D = *dotprecision*-Variable,
C,C1,C2 = *complex*-Ausdruck, LC,LC1,LC2 = *Lcomplex*-Ausdruck,
DC,DC1,DC2 = *cdotprecision*-Variable, DCI = *cidotprecision*-Variable.

6.2.2.2 Wertzuweisungen

Durch Überladen des Zuweisungsoperators := sind folgende Wertzuweisungen an eine Variable vom Typ *cidotprecision* möglich:

cidotprecision–Variable	:=	integer/real–Ausdruck;
cidotprecision–Variable	:=	Longreal–Ausdruck;
cidotprecision–Variable	:=	dotprecision–Ausdruck;
cidotprecision–Variable	:=	complex–Ausdruck;
cidotprecision–Variable	:=	Lcomplex–Ausdruck;
cidotprecision–Variable	:=	cdotprecision–Ausdruck;
cidotprecision–Variable	:=	interval-Ausdruck;
cidotprecision–Variable	:=	Linterval–Ausdruck;
cidotprecision–Variable	:=	idotprecision–Ausdruck;
cidotprecision–Variable	:=	cinterval–Ausdruck;
cidotprecision–Variable	:=	Lcinterval–Ausdruck;

6.2.2.3 Operatoren

Auch jetzt muß z.B. für einen Multiplikationsoperator mit *Lcinterval*-Operanden und einem Ergebnis vom Typ *cidotprecision* ein **neuer** Operatorname (times_ci) gewählt werden, da der normale $*$ Operator mit gleichen Operanden schon für den Ergebnistyp *Lcinterval* vergeben ist.
Entsprechende Überlegungen bezüglich der Additions- und Subtraktionsoperatoren (z.B mit Namen plus_ci, minus_ci) führen zum Ergebnis, daß bei Berücksichtigung aller Operandenkombinationen neben den 81 times_ci Operatoren noch zusätzlich $2 \cdot 13 \cdot 13 = 338$ Operatoren zu definieren sind! Die große Zahl von insgesamt 419 Operatoren läßt sich auf $81 + 2 \cdot 13 = 107$ reduzieren, wenn man bei Addition und Subtraktion die folgende für die Praxis unerhebliche Einschränkung bezüglich der Operandenkombinationen macht. Dazu betrachten wir die Variable S vom Typ *cidotprecision* und die Variablen a,b,c mit den möglichen Typen:

> cidotprecision, Lcinterval, cinterval, cdotprecision, Lcomplex, complex, dotprecision, Longreal, real, integer, idotprecision, Linterval, interval;

Da für den Compiler die beiden folgenden Anweisungen äquivalent sind

$$\mathrm{S} := \mathrm{a} + \mathrm{b} + \mathrm{c} \quad \Longleftrightarrow \quad \mathrm{S} := (\mathrm{a} + \mathrm{b}) + c,$$

kann man mit der Variablen NULL_DCI: *cidotprecision* die Anweisung $\mathrm{S} := \mathrm{a}+\mathrm{b}$ auch in der Form schreiben:

$$\begin{aligned} \mathrm{NULL\_DCI} &:= 0; \\ \mathrm{S} &:= \mathrm{NULL\_DCI} + \mathrm{a} + \mathrm{b} \quad \Longleftrightarrow \quad \mathrm{S} := ((\mathrm{NULL\_DCI} + \mathrm{a}) + \mathrm{b}), \end{aligned}$$

und jetzt werden in der Anweisung für S **nur** Additionsoperatoren mit einem linken Operanden vom Typ *cidotprecision* benutzt! Für eine Subtraktion $\mathrm{S} := \mathrm{a} - \mathrm{b}$ gilt dann entsprechend:

$$\mathrm{NULL\_DCI} := 0; \qquad \mathrm{S} := \mathrm{NULL\_DCI} + \mathrm{a} - \mathrm{b};$$

In den nachfolgenden Tabellen 6.13, 6.14 sind alle Operanden der Operatoren aus dem Modul LCI_ARI mit Ergebnistyp *cidotprecision* zusammengestellt:

rechter Operand	real Longreal	interval Linterval	complex Lcomplex	cinterval Lcinterval
linker Operand				
cidotprecision	$\overset{*}{+,-}$	$\overset{*}{+,-}$	$\overset{*}{+,-}$	$\overset{*}{+,-}$
real Longreal	times_ci	times_ci	times_ci	times_ci
interval Linterval	times_ci	times_ci	times_ci	times_ci
complex Lcomplex	times_ci	times_ci	times_ci	times_ci
cinterval Lcinterval	times_ci	times_ci	times_ci	times_ci

Tabelle 6.13: Operatoren aus LCI_ARI mit Ergebnistyp *cidotprecision* (Teil 1)

rechter Operand	dotprecision idotprecision	cdotprecision	cidotprecision
linker Operand			
cidotprecision	$\overset{*}{+,-}$	$\overset{*}{+,-}$	$\overset{*}{+,-}$
real Longreal			*
interval Linterval			*
complex Lcomplex			*
cinterval Lcinterval			*
dotprecision idotprecision cdotprecision			*

Tabelle 6.14: Operatoren aus LCI_ARI mit Ergebnistyp *cidotprecision* (Teil 2)

Beachten Sie, daß jetzt neue Namen für Additions- und Subtraktionsoperatoren **nicht** notwendig sind, da die Einschränkung besteht, daß für diese Operatoren mit Ergebnistyp *cidotprecision* der linke Operand vom Typ *cidotprecision* sein muß. Durch diese Einschränkung ergibt sich noch der wichtige Satz:

> In einem Ausdruck mit Ergebnistyp *cidotprecision* dürfen keine Klammern gesetzt werden.

Aus den Tabellen 6.13, 6.14 ergibt sich noch:

Ist bei einer Multiplikation mit Ergebnistyp *cidotprecision* wenigstens ein Operand vom Typ *cidotprecision*, so wird zur Vereinfachung das übliche Operatorsymbol $*$ benutzt und nicht der Name times_ci.

Beispiele:

1. Sind a,b vom Typ *real* und S vom Typ *cidotprecision*, so ist die Anweisung

   $$\mathrm{S} := \mathrm{b} + (\mathrm{a} - \mathrm{b})$$

   zwar zulässig, aber die Subtraktion in der Klammer und die anschließende Addition liefern i.a. fehlerbehaftete *real*-Ergebnisse, so daß in S i.a. **nicht** der exakte Differenzwert a gespeichert wird! Mit den Anweisungen

   $$\mathrm{NULL\_DCI} := 0; \qquad \mathrm{S} := \mathrm{NULL\_DCI} + \mathrm{b} + \mathrm{a} - \mathrm{b}$$

   erhält man jedoch ohne Klammern den **exakten** Differenzwert a.

2. Sind a,b,c vom Typ *real* und S vom Typ *cidotprecision*, so ist die Anweisung

   $$\mathrm{S} := \mathrm{c} * (\mathrm{a} - \mathrm{b})$$

   ebenfalls zulässig, liefert aber wie im 1. Beispiel i.a. **nicht** das exakte Produkt! Mit der Anweisung

   $$\mathrm{S} := \mathrm{c}\ \mathrm{times\_ci}\ \mathrm{a}\ -\ \mathrm{c}\ \mathrm{times\_ci}\ \mathrm{b}$$

   erhält man jedoch ohne Klammern das **exakte** Produkt c$\cdot(a-b)$.
   Natürlich liefert die Anweisung

   $$\mathrm{S} := \#(c \cdot a - c \cdot b);$$

   ebenfalls das exakte Produkt $c \cdot (a-b)$ mit dem Ergebnistyp *cidotprecision*.

6.2.2.4 Simulation von SUM–Ausdrücken

Zur Simulation von **SUM**–Ausdrücken im ##–Konzept stehen die folgenden Datentypen und Prozeduren zur Verfügung:

- **type** civector = **dynamic** array [*] of **cinterval**;
- **type** Lcivector = **dynamic** array [*] of **Lcinterval**;

1. procedure **ADDNACCU** (var dci: cidotprecision; var a: Lcivector);
   Exakte Addition des Lcinterval-Feldes a zur *cidotprecision*-Variablen dci;
2. procedure **ADDNACCU** (var dci: cidotprecision; var a: civector);
   Exakte Addition des *cinterval*-Feldes a zur *cidotprecision*-Variablen dci;
3. procedure **ADDNACCU** (var dci: cidotprecision; var a: Livector);
   Exakte Addition des *Linterval*-Feldes a zur *cidotprecision*-Variablen dci;
4. procedure **ADDNACCU** (var dci: cidotprecision; var a: ivector);
   Exakte Addition des *interval*-Feldes a zur *cidotprecision*-Variablen dci;
5. procedure **ADDNACCU** (var dci: cidotprecision; var a: Lcvector);
   Exakte Addition des *Lcomplex*-Feldes a zur *cidotprecision*-Variablen dci;
6. procedure **ADDNACCU** (var dci: cidotprecision; var a: cvector);
   Exakte Addition des *complex*-Feldes a zur *cidotprecision*-Variablen dci;
7. procedure **ADDNACCU** (var dci: cidotprecision; var a: Lrvector);
   Exakte Addition des *Longreal*-Feldes a zur *cidotprecision*-Variablen dci;
8. procedure **ADDNACCU** (var dci: cidotprecision; var a: rvector);
   Exakte Addition des *real*-Feldes a zur *cidotprecision*-Variablen dci;
9. procedure **SUBNACCU** (var dci: cidotprecision; var a: Lcivector);
   Exakte Subtraktion des Lcinterval-Feldes a zur *cidotprecision*-Variablen dci;
10. procedure **SUBNACCU** (var dci: cidotprecision; var a: civector);
    Exakte Subtraktion des *cinterval*-Feldes a zur *cidotprecision*-Variablen dci;
11. procedure **SUBNACCU** (var dci: cidotprecision; var a: Livector);
    Exakte Subtraktion des *Linterval*-Feldes a zur *cidotprecision*-Variablen dci;
12. procedure **SUBNACCU** (var dci: cidotprecision; var a: ivector);
    Exakte Subtraktion des *interval*-Feldes a zur *cidotprecision*-Variablen dci;
13. procedure **SUBNACCU** (var dci: cidotprecision; var a: Lcvector);
    Exakte Subtraktion des *Lcomplex*-Feldes a zur *cidotprecision*-Variablen dci;
14. procedure **SUBNACCU** (var dci: cidotprecision; var a: cvector);
    Exakte Subtraktion des *complex*-Feldes a zur *cidotprecision*-Variablen dci;

15. procedure **SUBNACCU** (var dci: cidotprecision; var a: Lrvector);
Exakte Subtraktion des *Longreal*-Feldes a zur *cidotprecision*-Variablen dci;

16. procedure **SUBNACCU** (var dci: cidotprecision; var a: rvector);
Exakte Subtraktion des *real*-Feldes a zur *cidotprecision*-Variablen dci;

17. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: Lcivector);
Exakte Addition des *Lcinterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

18. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: civector);
Exakte Addition des *cinterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

19. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: Livector);
Exakte Addition des *Linterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

20. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: ivector);
Exakte Addition des *interval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

21. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: Lcvector);
Exakte Addition des *Lcomplex*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

22. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: cvector);
Exakte Addition des *complex*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

23. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: Lrvector);
Exakte Addition des *Longreal*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

24. procedure **PADDNACCU** (var dci: cidotprecision; var a,b: rvector);
Exakte Addition des *real*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

25. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: Lcivector);
Exakte Subtraktion des *Lcinterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

26. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: civector);
Exakte Subtraktion des *cinterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

27. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: Livector);
Exakte Subtraktion des *Linterval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

28. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: ivector);
Exakte Subtraktion des *interval*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

29. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: Lcvector);
Exakte Subtraktion des *Lcomplex*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

30. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: cvector);
Exakte Subtraktion des *complex*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

31. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: Lrvector);
Exakte Subtraktion des *Longreal*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

32. procedure **PSUBNACCU** (var dci: cidotprecision; var a,b: rvector);
Exakte Subtraktion des *real*-Skalarproduktes zur Variablen dci vom Typ *cidotprecision*;

6.2.2.5 Beispiele

1. Beispiel:

Mit den Deklarationen:

```
var dotci,dci : cidotprecision;
    a,b       : Lcivector [1..10];
    Lci       : Lcinterval;
```

sind z.B. folgende Anweisungen möglich:

```
dci := 0;
PADDNACCU (dci,a,b);
dotci := dci + a[1] times_ci b[10] − a[2] times_ci b[9];
Lci := LONG (dotci);
```

Das komplexe Intervall Lci : Lcinterval liefert eine Einschließung der exakten, komplexen Intervallsumme dotci mit **nur einer Rundung** bezüglich der reellen bzw. imaginären Intervallunter– und –Obergrenze!

2. Beispiel:

Mit Hilfe der Operatoren $*$ und $+$ zwischen Operanden vom Typ *cidotprecision* und *Lcinterval* kann ein Polynom vom Ergebnistyp *cidotprecision* mit Argument und Koeffizienten vom Typ *Lcinterval*

$$\mathrm{P}(z) = \sum_{k=0}^{n} a_k \cdot z^k$$

mit der folgenden Funktion nach dem Hornerschema **rundungsfehlerfrei** berechnet werden:

```
function HORNER_CIDOT (var a: Lcivector;
                           z: Lcinterval): cidotprecision;

var k : integer;
    y : cidotprecision;

begin
    y := a[ub(a)];
    for k := ub(a)−1 downto lb(a) do
    y := y * z + a[k];
    HORNER_CIDOT := y
end;
```

Die komplexen Punktintervall-Koeffizienten

$$a_0 = 120 + 90i; \quad a_1 = -54 + 22i; \quad a_2 = 6 + 7i;$$
$$a_3 = -5 - 3i; \quad a_4 = 1 + 0i;$$

realisieren ein komplexwertiges Polynom P(z) mit den Nullstellen:

$$z_{1,2} = \pm\sqrt{2} \cdot (1 + 2i), \quad z_3 = 5, \quad z_4 = 3i.$$

Mit den Deklarationen:

```
var a: Lcivector[0..4];
    z,y: Lcinterval;
    dci: cidotprecision;
```

und den Anweisungen:

```
z     := 5.00000000000000000001;   { z ≈ z3 }
dci   := HORNER_CIDOT (a,z);
y     := LONG(dci);
```

erhält man mit der BCD–Version das Ergebnis:

$$y = \mathrm{P}(z) \in (\ [+1.3100\ldots000 \cdot 10^{-18}, +1.3100\ldots001 \cdot 10^{-18}],$$
$$[-1.3300\ldots001 \cdot 10^{-18}, -1.3300\ldots000 \cdot 10^{-18}]\ )$$

mit **nur einer Rundung** bzgl. Unter- und Obergrenze der Real- und Imaginärteilintervalle.

6.2.3 Komplexe Intervallrechnung mit dem Typ rrG

Mit dem Modul **LCI_ARI** werden nur diejenigen Operatoren, Funktionen und Prozeduren bereitgestellt, die zur Implementierung von komplexen Intervall–Standardfunktionen zum Basistyp rrG notwendig sind. Es ist daher nicht vorgesehen, die komplette Intervallrechnung, wie sie etwa zwischen den Typen *cinterval* und *Lcinterval* definiert wurde, ebenfalls zu realisieren, so daß z.B. Vergleichsoperatoren für Intervalle oder Intervall–Grundoperationen mit gemischten Operandentypen nur in dem Umfang bereitgestellt werden, wie sie tatsächlich erforderlich waren.

6.2.3.1 Der Datentyp rrGcinterval

Der Datentyp *rrGcinterval* ist definiert durch:

type rrGcinterval = **record** re, im : rrGinterval **end**;

Für z : *rrGcinterval* sind Real- und Imaginärteil von z **Intervalle**, so daß

$$z = z.re + i \cdot z.im = [z.re.inf, z.re.sup] + i \cdot [z.im.inf, z.im.sup]; \qquad i = \sqrt{-1}$$

in der komplexen Ebene als Rechteck-Intervall gedeutet werden kann:

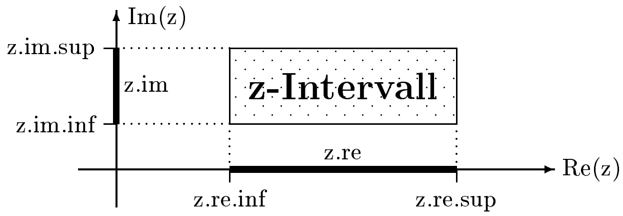


Abbildung 6.3: Komplexes Intervall

6.2.3.2 Transferfunktionen

Zur Wandlung zwischen den Typen *rrGcinterval* und *rrGinterval* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
RRGCINTVAL (x1,x2)	*rrGcinterval*	$z = x1 + i \cdot x2; \quad i = \sqrt{-1}$
RRGCINTVAL (x)	*rrGcinterval*	$z = x + i \cdot 0; \quad i = \sqrt{-1}$
RE (z)	*rrGinterval*	z.Re = Realteil von z
IM (z)	*rrGinterval*	z.Im = Imaginärteil von z

x, x1, x2 = *rrGinterval*–Ausdruck; z = *rrGcinterval*–Ausdruck;

6.2.3.3 Operatoren

Neben den monadischen Operatoren $+ \; -$ existieren die in folgender Tabelle zusammengestellten, arithmetischen Grundoperatoren:

1. Operand		**2. Operand**	**Ergebnistyp**
rrGcinterval	$+$	rrGcinterval	rrGcinterval
rrGcinterval	$-$	rrGcinterval	rrGcinterval
rrGcinterval	$*$	rrGcinterval	rrGcinterval
real	$*$	rrGcinterval	rrGcinterval
Longreal	$*$	rrGcinterval	rrGcinterval
rrG	$*$	rrGcinterval	rrGcinterval
rrGcinterval	$/$	rrGcinterval	rrGcinterval
rrGcinterval	$/$	real	rrGcinterval
rrGcinterval	$/$	Longreal	rrGcinterval
rrGcinterval	$/$	rrG	rrGcinterval

Als Vergleichsoperatoren existieren lediglich die Operatoren $= \; <>$.

6.2.3.4 Überladungen des Zuweisungsoperators

Die Wandlungen zwischen den verschiedenen Datentypen werden in Form überladener Zuweisungen nach folgender Tabelle bereitgestellt:

Zuweisung	**Bedeutung**
rrGci := R	rrGci = reelle Zahl
rrGci := C	rrGci = komplexe Zahl
rrGci := RI	rrGci = reelles Intervall
rrGci := CI	rrGci = komplexes Intervall
LC := rrGci	rrGci $\subseteq$ LC

Tabelle 6.15: Überladungen des Zuweisungsoperators

rrGci : rrGcinterval; R : *rrG/Longreal/real*; C : *rrGcomplex/Lcomplex/complex*; RI : *rrGinterval/Linterval/interval*; CI : *Lcinterval/cinterval*; LC : *Lcinterval*;

6.2.3.5 Standardfunktionen

Es stehen folgende Standardfunktionen mit Argumenten x,y: rrGcinterval zur Verfügung:

Funktion	**Ergb.-Typ**	**Bedeutung**	
sqr (x)	rrGcinterval	$x^2 = x \cdot x$	Quadrat
sqrt (x)	rrGcinterval	$\sqrt{x}$	Quadratwurzel
exp (x)	rrGcinterval	e^x	Exponentialfunktion
ln (x)	rrGcinterval	$\ln(x)$	Natürlicher Logarithmus
ln1p (x)	rrGcinterval	$\ln(1+x)$	Natürlicher Logarithmus
power (x,y)	rrGcinterval	x^y	Potenzfunktion
xp1powy (x,y)	rrGcinterval	$(1+x)^y$	Potenzfunktion
sin (x)	rrGcinterval	$\sin(x)$	Sinus
cos (x)	rrGcinterval	$\cos(x)$	Kosinus
sinh (x)	rrGcinterval	$\sinh(x)$	Hyperbolischer Sinus
cosh (x)	rrGcinterval	$\cosh(x)$	Hyperbolischer Kosinus
conj (x)	rrGcinterval	$\overline{x} = u - i \cdot v$	Konjugation von $x = u + i \cdot v$
arg (x)	rrGinterval	φ	Argumentintervall für das x-Rechteck
arg1p (x)	rrGinterval	φ	Argumentintervall für das $(1+x)$-Rechteck
abs (x)	rrGinterval	$\sqrt{x.re^2 + x.im^2}$	Absolutbetrag von x
diam (x)	rrG	d	Durchmesser von x

Tabelle 6.16: Standardfunktionen des Moduls LCI_ARI

x,y = *rrGcinterval*-Ausdruck, $i = \sqrt{-1}$

Anmerkungen zu den Standardfunktionen:

1. Für die Funktionen **sqrt, arg, arg1p, ln, ln1p, power, xp1powy** vom Ergebnistyp *rrGcinterval* gelten bezüglich der Verzweigungsschnitte in Abhängigkeit von der Lage des komplexen Argumentintervalls die gleichen Aussagen wie für die entsprechenden *cinterval*-Funktionen von Seite 131.

2. d := diam (x) liefert zum komplexen Intervall x: *rrGcinterval* eine Oberschranke d: *rrG* für die Länge L der entsprechenden Rechteckdiagonalen mit: $0 \leq L \leq d$;

Es stehen folgende Standardfunktionen mit Argumenten x,y: Lcinterval zur Verfügung:

Funktion	Ergb.-Typ	Bedeutung	
sqr_rrG (x)	rrGcinterval	$x^2 = x \cdot x$	Quadrat
sqrt_rrG (x)	rrGcinterval	$\sqrt{x}$	Quadratwurzel
exp_rrG (x)	rrGcinterval	e^x	Exponentialfunktion
ln_rrG (x)	rrGcinterval	$\ln(x)$	Natürlicher Logarithmus
ln1p_rrG (x)	rrGcinterval	$\ln(1+x)$	Natürlicher Logarithmus
power_rrG (x,y)	rrGcinterval	x^y	Potenzfunktion
xp1powy_rrG (x,y)	rrGcinterval	$(1+x)^y$	Potenzfunktion
sin_rrG (x)	rrGcinterval	$\sin(x)$	Sinus
cos_rrG (x)	rrGcinterval	$\cos(x)$	Kosinus
sinh_rrG (x)	rrGcinterval	$\sinh(x)$	Hyperbolischer Sinus
cosh_rrG (x)	rrGcinterval	$\cosh(x)$	Hyperbolischer Kosinus
arg_rrG (x)	rrGinterval	φ	Argumentintervall für das x-Rechteck
arg1p_rrG (x)	rrGinterval	φ	Argumentintervall für das $(1+x)$-Rechteck
abs_rrG (x)	rrGinterval	$\sqrt{x.re^2 + x.im^2}$	Absolutbetrag von x
diam_rrG (x)	rrG	d	Durchmesser von x

Tabelle 6.17: Standardfunktionen des Moduls LCI_ARI

x,y = *Lcinterval*-Ausdruck, $i = \sqrt{-1}$

Anmerkungen zu den Standardfunktionen:

1. Für die Funktionen **sqrt_rrG, arg_rrG, arg1p_rrG, ln_rrG, ln1p_rrG, power_rrG, xp1powy_rrG** vom Ergebnistyp *rrGcinterval* gelten bezüglich der Verzweigungsschnitte in Abhängigkeit von der Lage des komplexen Argumentintervalls die gleichen Aussagen wie für die entsprechenden *cinterval*-Funktionen von Seite 131.

2. d := diam_rrG (x) liefert zum komplexen Intervall x: *Lcinterval* eine Oberschranke d: *rrG* für die Länge L der entsprechenden Rechteckdiagonalen mit: $0 \le L \le d$;

6.2.3.6 Wertebereich separierbarer Funktionen

Zur Berechnung einer Wertebereichs–Einschließung einer reellwertigen Funktion $u(z)$ mit komplexen Punktargumenten z dient die **PASCAL-XSC** Funktion

```
function BOUNDS (
         mi,Ma : rrGcomplex;  { Kompl. Punkte aus dem  }
                              { Def.-Bereich von  u(z) }
         function U(z: rrGcomplex): rrG; { reellwertig }
         eps : real { eps := 1.00001 *> eps(u)         }
                    { eps(u) := Fehlerschr.  von  u(z) }
         ) : rrGinterval;
```

$U(z)$ sei Näherungsfunktion für die reellwertige Funktion $u(z)$:

$$U(z) = u(z) \cdot (1 + \varepsilon_u), \quad |\varepsilon_u| < \varepsilon(u)$$

Der Eingabeparameter `eps` ist definiert durch:

$$\texttt{eps} := 1.00001 *\!> \varepsilon(u);$$

Mit der Anweisung

```
    y := BOUNDS( mi,Ma,U,eps );      { y: rrGinterval }
```

erhält man unter der Voraussetzung $u(\mathrm{mi}) \leq u(\mathrm{Ma})$ die folgenden Ergebnisse:

$$\mathrm{y.Inf} \leq u(\mathrm{mi}) \leq u(\mathrm{Ma}) \leq \mathrm{y.Sup}$$

Anmerkungen:

- Beim Aufruf von $U(z)$ muß in der globalen Variablen FW_exakt , die vom Standardmodul **STDMOD** bereitgestellt wird, festgehalten werden, ob $u(z)$ exakt berechnet wurde ($U(z) = u(z)$) oder nicht.
- Im Fall $u(\mathrm{mi}) \leq u(z) \leq u(\mathrm{Ma})$; $z \in \mathrm{D}$: rrGcinterval liefert y: rrGinterval eine Wertebereichseinschließung bzgl. $u(z)$ für $z \in \mathrm{D}$.
  Ist $U(z)$ der Realteil einer gegebenen, komplexwertigen Näherungsfunktion

  $$F(z) \equiv U(z) + i \cdot V(z) \approx f(z) \equiv u(z) + i \cdot v(z)$$

  so liefert y eine Einschließung des Realteils $u(z)$. Ganz analog läßt sich dann auch der Wertebereich des Imaginärteils von $f(z)$ einschließen.
  Ist die komplexwertige Funktion $f(z) := u(z) + i \cdot v(z)$ separierbar und sind die Minimum- und Maximumpunkte mi,Ma jeweils für $u(z)$ und $v(z)$ bekannt, so kann man durch Anwendung der Funktion **BOUNDS** auf die reellwertigen Näherungsfunktionen $U(z)$, $V(z)$ Intervall–Einschließungen für Real- und Imaginärteil von $f(z)$ berechnen.

6.2.3.7 Ein-/Ausgabeprozeduren

Es stehen die bekannten Prozeduren

> **procedure** read (**var** f: text; **var** RRGCI: rrGcinterval);
> **procedure** write (**var** f: text; RRGCI: rrGcinterval);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Intervalls RRGCI $= [x, y] + i \cdot [v, w]$ kann mit dem Prozeduraufruf **read**(RRGCI) in der Form

$([x, y], [v, w])$ allgemeines komplexes Intervall

oder in der Form

$(x, [v, w])$ mit Punktintervall als Realteil, d.h. $x = y$

oder in der Form

$([x, y], v)$ mit Punktintervall als Imaginärteil, d.h. $v = w$

oder in der Form

$[x, y]$ rein reelles Intervall, d.h. $v = w = 0$

oder in der Form

(x, v) komplexes Punktintervall, d.h. $x = y$ und $v = w$

oder in der Form

x rein reelles Punktintervall, d.h. $x = y$ und $v = w = 0$

erfolgen.

Die Vorschriften zur Eingabe einer rrG–Zahl x findet man dabei auf Seite 56.

Die Ausgabe eines komplexen Intervalls RRGCI erfolgt durch den Prozeduraufruf **write**(RRGCI) stets in der Form

$$([x, y], [v, w])$$

Die Ausgabe der Intervallgrenzen x, y und v, w kann natürlich auch erfolgen durch:

write(RRGCI.re.inf, RRGCI.re.sup) und write(RRGCI.im.inf, RRGCI.im.sup);

Kapitel 7

Matrix/-Vektorarithmetik

7.1 Übersicht

Das Rechnen mit Matrizen und Vektoren bildet die Grundlage zur Lösung vieler Probleme, z.B.:

> lineare und nichtlineare Gleichungssysteme, lineare Optimierung, Polynomauswertungen, globale Optimierung, ...

Für Matrix- und Vektoroperanden stehen die Grundoperationen $+, -, *$ mit den drei verschiedenen Rundungsarten zur Verfügung. Mit Hilfe des Akkumulators können die Operationen maximalgenau durchgeführt werden, d.h. das wahre Ergebnis z.B. eines Matrixelements aus einem Matrizenprodukt $A*B$ wird zur nächstgelegenen Maschinenzahl gerundet. Bei gerichteten Rundungen wird das exakte Ergebnis zur nächstkleineren bzw. nächstgrößeren Rasterzahl gerundet. Erst durch solche maximalgenauen Rechnungen wird z.B. die maximalgenaue Einschließung eines Defekts bei linearen Gleichungssystemen oder die maximalgenaue Einschließung eines reellen Polynomwertes ermöglicht! Die genannten Grundoperationen beziehen sich dabei auf reelle oder komplexe Matrix/-Vektoroperanden in Punkt- oder Intervallform.

Im Gegensatz zu den Modulen

STDMOD, LI_ARI, LC_ARI oder LCI_ARI

steht bei Matrizenrechnungen entweder nur der Datentyp *real* oder nur der Datentyp *Longreal* zur Verfügung. Beachten Sie, daß in der vorliegenden BCD-Version die in der Binärversion oft lästigen Konversionsfehler bei Matrix- und Vektorkomponenten **nicht** auftreten!

Die nachfolgende Tabelle liefert eine Zusammenstellung der vorhandenen Module, in denen die für das Rechnen mit Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt werden.

Name	**Inhalt**	Basistyp
MV_ARI	Reelle Matrizen und Vektoren	*real*
MVI_ARI	Reelle Intervall-Matrizen und -Vektoren	*real*
MVC_ARI	Komplexe Matrizen und Vektoren	*real*
MVCI_ARI	Komplexe Intervall-Matrizen und -Vektoren	*real*
LMV_ARI	Reelle Matrizen und Vektoren	*Longreal*
LMVI_ARI	Reelle Intervall-Matrizen und -Vektoren	*Longreal*
LMVC_ARI	Komplexe Matrizen und Vektoren	*Longreal*
LMVCI_AR	Komplexe Intervall-Matrizen und -Vektoren	*Longreal*

Tabelle 7.1: Module zur Matrix/-Vektorarithmetik

Die Namen der Module zum Basistyp **Longreal** ergeben sich also aus den Namen der Module zum Basistyp **real** durch Voranstellen des Buchstaben 'L' , wie es auch bei den entsprechenden Typ–Namen gehandhabt wurde.

7.2 Das Modul MV_ARI

In diesem Modul werden die für das Rechnen mit reellen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **real** bereitgestellt.

7.2.1 Datentypen

Die dynamischen Datentypen zur Darstellung von Vektoren und Matrizen sind entsprechend der Vereinbarung

type rvector = **dynamic array** [*] **of** real;
rmatrix = **dynamic array** [*] **of** rvector;

im Sprachkern von PASCAL–XSC enthalten. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.2.2 Operatoren

Viele der bekannten, grundlegenden Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Vektoren und Matrizen komponentenweise erklärt sind gemäß

c	:=	a	±	b	mit	c [i]	:=	a [i] ± b [i]
C	:=	A	±	B	mit	C [i,j]	:=	A [i,j] ± B [i,j]

mit a, b, c vom Typ *rvector* und A, B, C vom Typ *rmatrix*, sind die Operationen $*$ und $/$ definiert durch

s	:=	a	*	b	mit	s	:=	#* (**for** i := lb(a) **to** ub(a) **sum** (a [i] * b [i])) ‡
c	:=	r	*	a	mit	c [i]	:=	r * a [i]
c	:=	a	*	r	mit	c [i]	:=	a [i] * r
c	:=	a	/	r	mit	c [i]	:=	a [i]/r
c	:=	A	*	b	mit	c [i]	:=	A [i] * b ‡
C	:=	r	*	A	mit	C [i,j]	:=	r * A [i,j]
C	:=	A	*	r	mit	C [i,j]	:=	A [i,j] * r
C	:=	A	/	r	mit	C [i,j]	:=	A [i,j]/r
C	:=	A	*	B	mit	C [i,j]	:=	A [i] * B [*, j] ‡

‡ Maximalgenaues Skalarprodukt mit Ergebnistyp *real*

wobei r, s vom Typ *real*, a, b, c vom Typ *rvector* und A, B, C vom Typ *rmatrix* sind. Die Operationen mit gerichteten Rundungen sind entsprechend definiert.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ wirken auf alle Komponenten eines Vektors oder einer Matrix. Für a, b vom Typ *rvector* und A,B vom Typ *rmatrix* gilt:

$$\begin{array}{llllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[\,i\,] & <= & \mathrm{b}[\,i\,] & \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[\,i,j\,] & <= & \mathrm{B}[\,i,j\,] & \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *real*.

rechter Operand	integer real	rvector	rmatrix
linker Operand			
monadisch		$+, -$	$+, -$
integer real		$*, *<, *>$	$*, *<, *>$
rvector	$*, *<, *>$ $/, /<, />$	$\circ$ $\vee$	
rmatrix	$*, *<, *>$ $/, /<, />$	$*, *<, *>$	$\circ$ $\vee$

Tabelle 7.2: Operatoren des Moduls MV_ARI

$\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>\} \quad \vee \in \{=, <>, <, <=, >, >=\}$

7.2.3 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *rvector*- und *rmatrix*-Variablen wird in Form überladener Zuweisungen bereitgestellt:

Zuweisung	**Bedeutung**
rv := r	rv $[\,j\,]$:= r ; j = lb(rv),... ,ub(rv)
rM := r	rM $[\,j,k\,]$:= r ; j = lb(rM,1),... ,ub(rM,1) k = lb(rM,2),... ,ub(rM,2)

Tabelle 7.3: r = *real*-Ausdruck, rv = *rvector*-**Variable**, rM = *rmatrix*-**Variable**

7.2.4 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors sowie die Funktion *transp* zur Berechnung der transponierten Matrix zur Verfügung.

Funktion	**Ergebnistyp**	**Bedeutung**
null (v)	*rvector*	Nullvektor mit dem aktuellen Indexbereich von v
vnull (n)	*rvector*	Nullvektor mit dem Indexbereich [1..n]
null (M)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen von M
null (M1,M2)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $M1 \cdot M2$
null (n)	*rmatrix*	Nullmatrix mit Indexbereich [1..n, 1..n]
null (n1,n2)	*rmatrix*	Nullmatrix mit Indexbereich [1..n1, 1..n2]
id (M)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von M
id (M1,M2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $M1 \cdot M2$
id (n)	*rmatrix*	Einheitsmatrix mit den Indexbereichen [1..n, 1..n]
id (n1,n2)	*rmatrix*	Einheitsmatrix mit den Indexbereichen [1..n1, 1..n2]
transp (M)	*rmatrix*	Transponierte Matrix Mt von M mit $Mt\,[\,i,j\,] = M\,[\,j,i\,]$

Tabelle 7.4: n, n1, n2 = *integer*-Ausdruck, v = *rvector*-Ausdruck,
M, M1, M2 = *rmatrix*-Ausdruck

Beispiel:

Ist E die Einheitsmatrix und $R \approx A^{-1}$ eine Näherungsinverse der quadratischen Matrix A, so kann der in der Numerik häufig verwendete Defekt

$$D = E - R \cdot A$$

in PASCAL–XSC durch die Anweisung

$$D := \text{ id } (A) - R * A$$

bzw. unter Verwendung eines Lattenkreuzausdrucks durch die Anweisung

$$D := \# * \text{ (id } (A) - R * A)$$

bestimmt werden, wobei die zweite Form die Defektmatrix mit nur einer einzigen Rundung pro Komponente berechnet.

7.2.5 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : rvector);
procedure read (**var** f : text; **var** A : rmatrix);
procedure write (**var** f : text; a : rvector);
procedure write (**var** f : text; A : rmatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Vektors bzw. einer Matrix erfolgt komponentenweise entsprechend der Eingabe von *real*-Werten, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines Vektors bzw. einer Matrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *real*-Größen aus Tabelle 3.4.

Beispiel:

Durch die Anweisung

```
read (b, A, x);  { b,x : rvector; A : rmatrix; }
```

können der Vektor b, die Matrix A und der Vektor x hintereinander eingelesen werden.

7.3 Das Modul LMV_ARI

In diesem Modul werden die für das Rechnen mit reellen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **Longreal** bereitgestellt.

7.3.1 Datentypen

Die dynamischen Datentypen zur Darstellung von Vektoren und Matrizen sind entsprechend der Vereinbarung

type Lrvector = **dynamic array** [*] **of** Longreal;
Lrmatrix = **dynamic array** [*] **of** Lrvector;

definiert. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.3.2 Operatoren

Viele der bekannten, grundlegenden Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Vektoren und Matrizen komponentenweise erklärt sind gemäß

c	:=	a	$\pm$	b	mit	c [i]	:=	a [i] $\pm$ b [i]
C	:=	A	$\pm$	B	mit	C [i,j]	:=	A [i,j] $\pm$ B [i,j]

mit a,b,c vom Typ *Lrvector* und A, B, C vom Typ *Lrmatrix*, sind die Operationen $*$, *times* und / definiert durch

s	:=	a	$*$	b	mit	s	:=	Longnext ($\sum_i a[i] \cdot b[i]$) ‡
d	:=	a	*times*	g	mit	d	:=	$\sum_i a[i] \cdot g[i]$ †
c	:=	r	$*$	a	mit	c [i]	:=	r $*$ a [i]
c	:=	a	$*$	r	mit	c [i]	:=	a [i] $*$ r
c	:=	a	/	r	mit	c [i]	:=	a [i]/r
c	:=	A	$*$	b	mit	c [i]	:=	A [i] $*$ b ‡
C	:=	r	$*$	A	mit	C [i,j]	:=	r $*$ A [i,j]
C	:=	A	$*$	r	mit	C [i,j]	:=	A [i,j] $*$ r
C	:=	A	/	r	mit	C [i,j]	:=	A [i,j]/r
C	:=	A	$*$	B	mit	C [i,j]	:=	A [i] $*$ B[$*$, j] ‡

‡Maximalgenaues Skalarprodukt mit Ergebnistyp *Longreal*
†Exaktes Skalarprodukt mit Ergebnistyp *dotprecision*

wobei *r, s* vom Typ *Longreal*, *a, b, c* vom Typ *Lrvector*, *g* vom Typ *rvector/Lrvector*, *d* vom Typ *dotprecision* und *A, B, C* vom Typ *Lrmatrix* sind. Die Operationen mit gerichteten Rundungen sind ganz entsprechend definiert.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ wirken auf alle Komponenten eines Vektors oder einer Matrix. Für a, b vom Typ *Lrvector* und *A,B* vom Typ *Lrmatrix* gilt:

a	<=	b	$\Longleftrightarrow$	a[i]	<=	b[i]	für alle i
A	<=	B	$\Longleftrightarrow$	A[i,j]	<=	B[i,j]	für alle i,j

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *Longreal*.

rechter Operand	Longreal	rvector	Lrvector	Lrmatrix
linker Operand				
monadisch			$+, -$	$+, -$
Longreal			$*, *<, *>$	$*, *<, *>$
rvector		times	times	
Lrvector	$*, *<, *>$ $/, /<, />$	times	$\circ$, times $\vee$	
Lrmatrix	$*, *<, *>$ $/, /<, />$		$*, *<, *>$	$\circ$ $\vee$

Tabelle 7.5: Operatoren des Moduls LMV_ARI

$\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>\}$ $\quad \vee \in \{=, <>, <, <=, >, >=\}$

7.3.3 Überladungen des Zuweisungsoperators

Zuweisung			**Bedeutung**			
Lv	:=	Lr	Lv[j]	:=	Lr ;	j = lb(Lv),...ub(Lv)
LM	:=	Lr	LM[j,k]	:=	Lr ;	j = lb(LM,1),...ub(LM,1) k = lb(LM,2),...ub(LM,2)
LM	:=	M	LM[j,k]	:=	M[j,k] ;	j = lb(LM,1),...ub(LM,1) k = lb(LM,2),...ub(LM,2)
Lv	:=	v	Lv[j]	:=	v[j] ;	j = lb(Lv),...ub(Lv)

Lr = *Longreal*-Ausdruck, Lv = *Lrvector*-**Variable**,
LM = *Lrmatrix*-**Variable**, M = *rmatrix*-Ausdruck, v = *rvector*-Ausdruck

7.3.4 Standardfunktionen

Es stehen die Funktionen *lid* und *lnull* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors sowie die Funktion *ltransp* zur Berechnung der transponierten Matrix zur Verfügung. Die *short*-Funktionen erzeugen *rmatrix*- bzw. *rvector*-Werte.

Funktion	**Ergebnistyp**	**Bedeutung**
lnull (lv)	*Lrvector*	Nullvektor mit dem aktuellen Indexbereich von lv
lvnull (n)	*Lrvector*	Nullvektor mit dem Indexbereich [1..n]
lnull (lM)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen von lM
lnull (lM1,lM2)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $lM1 \cdot lM2$
lnull (n)	*Lrmatrix*	Nullmatrix mit Indexbereich [1..n, 1..n]
lnull (n1,n2)	*Lrmatrix*	Nullmatrix mit Indexbereich [1..n1, 1..n2]
lid (lM)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von lM
lid (lM1,lM2)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $lM1 \cdot lM2$
lid (n)	*Lrmatrix*	Einheitsmatrix mit den Indexbereichen [1..n, 1..n]
lid (n1,n2)	*Lrmatrix*	Einheitsmatrix mit den Indexbereichen [1..n1, 1..n2]
ltransp (lM)	*Lrmatrix*	Transponierte Matrix lMt von lM mit $lMt[i,j] = lM[j,i]$
short (lM)	*rmatrix*	$M[i,j] := short(lM[i,j])$
shortup (lM)	*rmatrix*	$M[i,j] := shortup(lM[i,j])$
shortdown (lM)	*rmatrix*	$M[i,j] := shortdown(lM[i,j])$
short (lv)	*rvector*	$v[j] := short(lv[j])$
shortup (lv)	*rvector*	$v[j] := shortup(lv[j])$
shortdown (lv)	*rvector*	$v[j] := shortdown(lv[j])$

Standardfunktionen des Moduls LMV_ARI, Teil 1
n, n1, n2 = *integer*-Ausdruck, lv = *Lrvector*-Ausdruck, v = *rvector*-**Variable**
M = *rmatrix*-**Variable**; lM, lM1, lM2 = *Lrmatrix*-Ausdruck;

Funktion	Typ	Bedeutung
short_test (lM,Err)	*rmatrix*	$M[i,j] := short\_test(lM[i,j], Err)$
shortup_test (lM,Err)	*rmatrix*	$M[i,j] := shortup\_test(lM[i,j], Err)$
shortdown_test (lM,Err)	*rmatrix*	$M[i,j] := shortdown\_test(lM[i,j], Err)$
short_test (lv,Err)	*rvector*	$v[j] := short\_test(lv[j], Err)$
shortup_test (lv,Err)	*rvector*	$v[j] := shortup\_test(lv[j], Err)$
shortdown_test (lv,Err)	*rvector*	$v[j] := shortdown\_test(lv[j], Err)$

Tabelle 7.6: Standardfunktionen des Moduls LMV_ARI, Teil 2

n, n1, n2 = *integer*-Ausdruck, lv = *Lrvector*-Ausdruck, v = *rvector*-**Variable**
M = *rmatrix*-**Variable**; lM, lM1, lM2 = *Lrmatrix*-Ausdruck;
Err = *boolean*-Variable

7.3.5 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : Lrvector);
procedure read (**var** f : text; **var** A : Lrmatrix);
procedure write (**var** f : text; a : Lrvector);
procedure write (**var** f : text; A : Lrmatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Vektors bzw. einer Matrix erfolgt komponentenweise entsprechend der Eingabe von *Longreal*-Werten, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines Vektors bzw. einer Matrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *Longreal*-Größen aus Tabelle 3.4.

Beispiel:

Durch die Anweisung

```
read (b, A, x);  { b,x : Lrvector; A : Lrmatrix }
```

können der Vektor b, die Matrix A und der Vektor x hintereinander eingelesen werden.

7.3.6 Exakte Auswertung von Ausdrücken

Mit Hilfe des Moduls **MV_ARI** lassen sich *rvector*-und *rmatrix*-Ausdrücke mit den drei verschiedenen Rundungsarten **maximalgenau** auswerten. So liefert z.B. die Anweisung

$$D := \# * (id(A) - R * A)$$

die Defektmatrix D vom Typ *rmatrix*, wobei bzgl. jeder Matrixkomponente nur eine einzige Rundung zur nächsten Rasterzahl erfolgt, vergl. Seite 163.

Mit a, rv vom Typ *rvector* und B vom Typ *rmatrix* liefert die Anweisung

$$rv := \# < (3 * a - B * a)$$

für die Variable rv bzgl. jeder Komponente den zur nächstkleineren *real*-Zahl gerundeten **exakten** Vektorausdruck $3 \cdot a - B \cdot a$, wobei $B \cdot a$ das bekannte Produkt einer Matrix B mit einem Spaltenvektor a bedeutet.

Da die obigen #-Ausdrücke nicht anwendbar sind, wenn nur einer der Operanden vom Typ *Longreal*, *Lrvector* oder *Lrmatrix* ist, soll die maximalgenaue Auswertung von *Lrvector*- und *Lrmatrix*-Ausdrücken simuliert werden! Aus Speicherplatzgründen lassen sich aber jetzt keine Operatoren für *Lrmatrix*-Operanden definieren, wenn für jede Ergebniskomponente eine *dotprecision*-Variable erforderlich wird. Man muß vielmehr zur Auswertung eines *Lrmatrix*-Ausdrucks jede Ergebniskomponente **einzeln** in einer *dotprecision*-Variablen rundungsfehlerfrei berechnen und ihren Wert dann anschließend mit einer der drei verschiedenen Rundungsarten in die entsprechende Ergebnis-Komponente vom Typ *Longreal* abspeichern. Dadurch benötigt man für die Auswertung eines beliebigen *Lrmatrix*-Ausdrucks nur eine einzige *dotprecision*-Variable; man muß jedoch die Ausdruckauswertung in einer Laufanweisung programmieren. Neben den Operatoren $plus, minus, times$ aus dem Standardmodul STDMOD benötigt man dazu aus dem Modul LMV_ARI lediglich die vier zusätzlichen Operatoren *times* mit *Lrvector*/*rvector*-Operanden und dem Ergebnistyp *dotprecision*:

OPERATOR **times** (var a,b: Lrvector) res: dotprecision;
OPERATOR **times** (var a,b: rvector) res: dotprecision;
OPERATOR **times** (var a: Lrvector; var b: rvector) res: dotprecision;
OPERATOR **times** (var a: rvector; var b: Lrvector) res: dotprecision;

Mit diesen Operatoren lassen sich dann auch *Lrmatrix*-Ausdrücke maximalgenau auswerten, die zusätzlich noch **r***matrix*-Operanden enthalten. Für die maximalgenaue Auswertung eines *Lrvector*-Ausdrucks gelten ganz entsprechende Aussagen. Die folgenden Programmbeispiele zeigen die typischen Vorgehensweisen bei der maximalgenauen Auswertung von *Lrvector*- und *Lrmatrix*-Ausdrücken.

```
program Beispiel1 (input, output);

{*  Ist R ≈ A^-1 eine Näherungsinverse, so liefert die Matrix D: Lrmatrix   *}
{*  den maximalgenauen Wert des Defekts   E − R · A ≈ D   , wobei E         *}
{*  die Einheitsmatrix mit dem Indexbereich von A ist.                       *}

use lmv_ari;
var E, R, A, D  : Lrmatrix [1..3, 1..3];
    i, k        : integer;
    Akku        : dotprecision;
```

```
begin
   E := lid (A);    { E = Einheitsmatrix mit Indexbereich von A }
   For i := lb(E,1) to ub(E,1) do
     For k := lb(E,2) to ub(E,2) do
     begin
        Akku := E[i,k] - R[i] times LRVECTOR( A[*,k] );
        D[i,k] := LONGNEXT( Akku );
     end;
end.
```

Anmerkungen:

1. Die Konstruktion LRVECTOR (A [∗, k]) liefert dem Compiler die Information, das Teilfeld A [∗, k] als Vektor vom Typ *Lrvector* zu behandeln.

2. Um die Matrix D maximalgenau berechnen zu können, müssen alle Operatoren in der Anweisung für die *dotprecision*-Variable Akku vom Ergebnistyp *dotprecision* sein!

3. Der Operator times mit Ergebnistyp *dotprecision* und *Lrvector*-Operanden ist im Modul LMV_ARI definiert; vergleiche Tabelle 7.5

4. Der Operator − mit *Longreal*- und *dotprecision*- Operanden liefert den Ergebnistyp *dotprecision* und könnte auch durch den Operator minus ersetzt werden. Beide Operatoren sind im Modul STDMOD definiert, vergl. Tabelle 3.9

Im nächsten Programmbeispiel wird mit a, lv vom Typ *Lrvector* und B vom Typ *rmatrix* der *Lrvector*-Ausdruck

$$3 \cdot a - B \cdot a$$

maximalgenau ausgewertet, wobei das exakte Ergebnis komponentenweise zur nächstkleineren *Longreal*-Zahl gerundet und in die Variable lv geschrieben wird.

```
program Beispiel2 (input, output);
use lmv_ari;
var B         : rmatrix [1..3, 1..3];
    i         : integer;
    a, lv     : Lrvector [1..3];
    Akku      : dotprecision;
begin
   For i := lb(a) to ub(a) do
     begin
        Akku := 3 times a[i] minus B[i] times a;
        lv[i] := LONGDOWN( Akku );
     end;
end.
```

Anmerkungen:

1. Um den Vektor lv maximalgenau berechnen zu können, müssen alle Operatoren in der Anweisung für die *dotprecision*-Variable Akku vom Ergebnistyp *dotprecision* sein!

2. Der erste Operator times mit *integer*/*Longreal*-Operanden und der Operator minus mit *dotprecision*-Operanden haben den gemeinsamen Ergebnistyp *dotprecision*; vergl. Tabelle 3.9

3. Der zweite Operator times mit *rvector*/*Lrvector*-Operanden hat den Ergebnistyp *dotprecision*; vergl. Tabelle 7.5 und Seite 169.

7.4 Das Modul MVI_ARI

In diesem Modul werden die für das Rechnen mit reellen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **real** bereitgestellt.

7.4.1 Datentypen

Die dynamischen Datentypen zur Darstellung von Intervallvektoren und Intervallmatrizen sind entsprechend der Vereinbarung

type ivector = **dynamic array** [*] **of** interval;
imatrix = **dynamic array** [*] **of** ivector;

im Sprachkern von PASCAL–XSC enthalten. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.4.2 Operatoren

Viele der bekannten, grundlegenden Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der komponentenweisen Rundung zum kleinsten einschließenden Intervall (auch für gemischte Typen) zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Intervallvektoren und Intervallmatrizen komponentenweise erklärt sind gemäß

c	:=	a	±	b	mit	c [i]	:= a [i] ± b [i]
C	:=	A	±	B	mit	C [i,j]	:= A [i,j] ± B [i,j]

mit a, b, c vom Typ *ivector* und A, B, C vom Typ *imatrix*, sind die Operationen $*$ und / definiert durch

s	:=	a	*	b	mit	s	:= ## (**for** i := lb(a) **to** ub(a) **sum** (a [i] * b [i])) [‡]
c	:=	r	*	a	mit	c [i]	:= r * a [i]
c	:=	a	*	r	mit	c [i]	:= a [i] * r
c	:=	a	/	r	mit	c [i]	:= a [i]/r
c	:=	A	*	b	mit	c [i]	:= A [i] * b [‡]
C	:=	r	*	A	mit	C [i,j]	:= r * A [i,j]
C	:=	A	*	r	mit	C [i,j]	:= A [i,j] * r
C	:=	A	/	r	mit	C [i,j]	:= A [i,j]/r
C	:=	A	*	B	mit	C [i,j]	:= A [i] * B [* , j] [‡]

[‡] Maximalgenaues Skalarprodukt mit Ergebnistyp *interval*

wobei *r, s* vom Typ *interval*, *a, b, c* vom Typ *ivector* und *A, B, C* vom Typ *imatrix* sind. Die Operationen mit gemischten Operandentypen sind entsprechend definiert.

rechter Operand	integer real	interval	rvector	ivector	rmatrix	imatrix
linker Operand						
monadisch				$+, -$		$+, -$
integer real				$*$		$*$
interval			$*$	$*$	$*$	$*$
rvector		$*, /$	$+*$	$\diamond$ $=, <>, \mathbf{in}$ $+*$		
ivector	$*, /$	$*, /$	$\diamond$ $=, <>$ $+*$	$\diamond$ $\mathbf{in}, \vee, ><$ $+*, **$		
rmatrix		$*, /$		$*$	$+*$	$\diamond$ $=, <>, \mathbf{in}$ $+*$
imatrix	$*, /$	$*, /$	$*$	$*$	$\diamond$ $=, <>$ $+*$	$\diamond$ $\mathbf{in}, \vee, ><$ $+*, **$

Tabelle 7.7: Die Operatoren des Moduls MVI_ARI

$$\vee \in \{=, <>, <, <=, >, >=\}\,; \qquad \diamond \in \{+, -, *\}$$

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind wie im Falle von Intervallen mengentheoretisch definiert und wirken auf alle Komponenten eines Intervallvektors oder einer Intervallmatrix. Für a, b vom Typ *ivector* und *A,B* vom Typ *imatrix* gilt:

$$\begin{array}{lllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[i] & <= & \mathrm{b}[i] & \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[i,j] & <= & \mathrm{B}[i,j] & \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *interval*.

Zusätzlich stehen noch die Operatoren **in** für die Relation „liegt in“ zwischen einem *rvector*- und einem *ivector*-Operanden bzw. zwischen einem *rmatrix*- und einem *imatrix*-Operanden und für die Relation „enthalten im Innern“ zwischen zwei *ivector*-Operanden bzw. *imatrix*-Operanden sowie >< für den Test auf Disjunktheit zweier Intervallvektoren bzw. Intervallmatrizen zur Verfügung. Diese Operatoren sind jeweils komponentenweise definiert. Die Verbandsoperatoren +* und ** bezeichnen die komponentenweise Bildung der Intervall-Hülle bzw. des Durchschnitts, wie es bereits für den Typ *interval* auf Seite 76 beschrieben wurde.

7.4.3 Transferfunktionen

Zur Wandlung zwischen den Typen *rvector* und *ivector* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
intval (rv1,rv2)	*ivector*	Intervallvektor iv mit $iv[\,i\,] = \text{intval}\ (rv1[\,i\,], rv2[\,i\,])$
intval (rv)	*ivector*	Punktintervallvektor iv mit $iv[\,i\,] = \text{intval}\ (rv[\,i\,])$
inf (iv)	*rvector*	Vektor rv der Untergrenzen mit $rv[\,i\,] = \text{inf}\ (iv[\,i\,])$
sup (iv)	*rvector*	Vektor rv der Obergrenzen mit $rv[\,i\,] = \text{sup}\ (iv[\,i\,])$

rv, rv1, rv2 = *rvector*-Ausdruck mit: rv1 <= rv2; iv = *ivector*-Ausdruck

Zur Wandlung zwischen den Typen *rmatrix* und *imatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
intval (rM1,rM2)	*imatrix*	Intervallmatrix iM mit $iM[\,i,j\,] = \text{intval}\ (rM1[\,i,j\,], rM2[\,i,j\,])$
intval (rM)	*imatrix*	Punktintervallmatrix iM mit $iM[\,i,j\,] = \text{intval}\ (rM[\,i,j\,])$
inf (iM)	*rmatrix*	Matrix rM der Untergrenzen mit $rM[\,i,j\,] = \text{inf}\ (iM[\,i,j\,])$
sup (iM)	*rmatrix*	Matrix rM der Obergrenzen mit $rM[\,i,j\,] = \text{sup}\ (iM[\,i,j\,])$

rM, rM1, rM2 = *rmatrix*-Ausdruck mit: rM1 <= rM2; iM = *imatrix*-Ausdruck

7.4.4 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *ivector*- und *imatrix*-Variablen sowie die Wandlung von *rvector* nach *ivector* und *rmatrix* nach *imatrix* werden in Form überladener Zuweisungen bereitgestellt:

Zuweisung			Bedeutung			
iv	:=	r	iv [j]	:=	intval (r) ;	j = lb(iv),... ,ub(iv)
iv	:=	i	iv [j]	:=	i ;	j = lb(iv),... ,ub(iv)
iv	:=	rv	iv	:=	intval (rv) ;	
iM	:=	r	iM [j,k]	:=	intval (r) ;	j = lb(iM,1),... ,ub(iM,1) k = lb(iM,2),... ,ub(iM,2)
iM	:=	i	iM [j,k]	:=	i ;	j = lb(iM,1),... ,ub(iM,1) k = lb(iM,2),... ,ub(iM,2)
iM	:=	rM	iM	:=	intval (rM) ;	

Tabelle 7.8: Überladungen des Zuweisungsoperators

i = *interval*-Ausdruck, iv = *ivector*-**Variable**, iM = *imatrix*-**Variable**
r = *real*-Ausdruck, rv = *rvector*-Ausdruck, rM = *rmatrix*-Ausdruck

7.4.5 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktionen *mid* und *diam* für die komponentenweise Berechnung von Mittelpunkt und Durchmesser sowie die Funktion *transp* zur Berechnung der transponierten Intervallmatrix zur Verfügung. Darüberhinaus wird die komponentenweise definierte Funktion *blow* für die Epsilonaufblähung bereitgestellt.

Funktion	Ergebnistyp	Bedeutung
null (iv)	*rvector*	Nullvektor mit dem aktuellen Indexbereich von *iv*
null (iM)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen von iM
null (iM1,iM2)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $iM1 \cdot iM2$

Funktion	Ergebnistyp	Bedeutung
id (iM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von iM
id (iM1,iM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $iM1 \cdot iM2$
mid (iv)	*rvector*	Mittelpunktsvektor pv mit $pv\,[\,i\,] = mid\,(iv\,[\,i\,])$
diam (iv)	*rvector*	Durchmesservektor dv mit $dv\,[\,i\,] = diam\,(iv\,[\,i\,])$
mid (iM)	*rmatrix*	Mittelpunktsmatrix pM mit $pM\,[\,i,j\,] = mid\,(iM\,[\,i,j\,])$
diam (iM)	*rmatrix*	Durchmessermatrix dM mit $dM\,[\,i,j\,] = diam\,(iM\,[\,i,j\,])$
transp (iM)	*imatrix*	Transponierte Matrix iMt von iM mit $iMt\,[\,i,j\,] = iM\,[\,j,i\,]$
blow (iv,r)	*ivector*	Vektorielle Epsilonaufblähung ev mit $ev\,[\,i\,] = blow\,(iv\,[\,i\,],r)$
blow (iM,r)	*imatrix*	Matrix-Epsilonaufblähung eM mit $eM\,[\,i,j\,] = blow\,(iM\,[\,i,j\,],r)$

Tabelle 7.9: Standardfunktionen des Moduls MVI_ARI

r = *real*-Ausdruck, iv = *ivector*-Ausdruck,
iM, iM1, iM2 = *imatrix*-Ausdruck

Beispiel:

A, D und R seien Variable vom Typ *imatrix*. A sei die quadratische Punktintervallmatrix der Punkt-Koeffizienten eines linearen Gleichungssystems. $R \approx A^{-1}$ sei die quadratische Punktintervallmatrix einer Näherungsinversen zu A. Dann läßt sich mit der Anweisung

$$D := id(A) - R * A;$$

eine garantierte Intervalleinschließung für den Defekt

$$DEF = E - R \cdot A, \qquad (E = \text{Einheitsmatrix})$$

berechnen, die wegen Intervallaufblähungen für die Praxis aber meist unbrauchbar ist!

Unter Verwendung eines Lattenkreuzausdrucks erhält man jedoch mit der Anweisung

$$D := \#\#(id(A) - R * A);$$

durch die Variable D die **engstmögliche** Einschließung für den Defekt DEF. In der letzten Anweisung können für R und A auch die entsprechenden Matrizen vom Typ *rmatrix* gewählt werden!

7.4.6 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : ivector);
procedure read (**var** f : text; **var** A : imatrix);
procedure write (**var** f : text; a : ivector);
procedure write (**var** f : text; A : imatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Intervallvektors bzw. einer Intervallmatrix erfolgt komponentenweise entsprechend der Eingabe von *interval*-Werten, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines Intervallvektors bzw. einer Intervallmatrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *real*-Größen aus Tabelle 3.4.

7.5 Das Modul LMVI_ARI

In diesem Modul werden die für das Rechnen mit reellen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **Longreal** bereitgestellt.

7.5.1 Datentypen

Die dynamischen Datentypen zur Darstellung von Intervallvektoren und Intervallmatrizen sind entsprechend der Vereinbarung

type Livector = **dynamic array** [∗] **of** Linterval;
Limatrix = **dynamic array** [∗] **of** Livector;

definiert. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.5.2 Operatoren

Viele der bekannten, grundlegenden Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der komponentenweisen Rundung zum kleinsten einschließenden Intervall (auch für gemischte Typen) zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Intervallvektoren und Intervallmatrizen komponentenweise erklärt sind gemäß

$$\begin{array}{ccccccccc} \mathrm{c} & := & \mathrm{a} & \pm & \mathrm{b} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] \pm \mathrm{b}[i] \\ \mathrm{C} & := & \mathrm{A} & \pm & \mathrm{B} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] \pm \mathrm{B}[i,j] \end{array}$$

mit a, b, c vom Typ *ivector* und A, B, C vom Typ *imatrix*, sind die Operationen $*$ und / definiert durch

$$\begin{array}{ccccccccl} \mathrm{s} & := & \mathrm{a} & * & \mathrm{b} & \text{mit} & \mathrm{s} & := & \sum_i a[i] \cdot b[i] \quad ^{\ddagger} \\ \mathrm{c} & := & \mathrm{r} & * & \mathrm{a} & \text{mit} & \mathrm{c}[i] & := & \mathrm{r} * \mathrm{a}[i] \\ \mathrm{c} & := & \mathrm{a} & * & \mathrm{r} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] * \mathrm{r} \\ \mathrm{c} & := & \mathrm{a} & / & \mathrm{r} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i]/\mathrm{r} \\ \mathrm{c} & := & \mathrm{A} & * & \mathrm{b} & \text{mit} & \mathrm{c}[i] & := & \mathrm{A}[i] * \mathrm{b} \quad ^{\ddagger} \\ \mathrm{C} & := & \mathrm{r} & * & \mathrm{A} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{r} * \mathrm{A}[i,j] \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{r} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] * \mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & / & \mathrm{r} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j]/\mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{B} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i] * \mathrm{B}[*,j] \quad ^{\ddagger} \end{array}$$

‡Maximalgenaues Skalarprodukt mit Ergebnistyp *Linterval*

wobei *r*, *s* vom Typ *Linterval*, *a*, *b*, *c* vom Typ *Livector* und *A*, *B*, *C* vom Typ *Limatrix* sind. Die Operationen mit gemischten Operandentypen sind entsprechend definiert.

rechter Operand	Longreal	Linterval	Lrvector	Livector	Lrmatrix	Limatrix
linker Operand						
monad.				$+, -$		$+, -$
Longreal				$*$		$*$
Linterval			$*$	$*$	$*$	$*$
Lrvector		$*, /$	$+*$	$\diamond$ $=, <>$,**in** $+*$		
Livector	$*, /$	$*, /$	$\diamond$ $=, <>$ $+*$	$\diamond$ **in**,$\vee, ><$ $+*, **$		
Lrmatrix		$*, /$		$*$	$+*$	$\diamond$ $=, <>$,**in** $+*$
Limatrix	$*, /$	$*, /$	$*$	$*$	$\diamond$ $=, <>$ $+*$	$\diamond$ **in**,$\vee, ><$ $+*, **$

Tabelle 7.10: Operatoren aus LMVI_ARI mit Ergebnistypen: *Linterval*, *Livector*, *Limatrix* oder *boolean*; $\vee \in \{=, <>, <, <=, >, >=\}$; $\diamond \in \{+, -, *\}$

rechter Operand	rvector	Lrvector	ivector	Livector
linker Operand				
rvector				times_i
Lrvector			times_i	times_i
ivector		times_i		
Livector	times_i	times_i		times_i

Tabelle 7.11: Operator times_i mit Ergebnistyp *idotprecision*

Zusätzlich zu den Operatoren aus Tabelle 7.10 steht zur exakten Auswertung von *Livector*- und *Limatrix*-Ausdrücken noch der Operator **times_i** mit dem Ergebnistyp *idotprecision* zur Verfügung. Die erlaubten Operandenkombinationen findet man in Tabelle 7.11.

Die Vergleichsoperatoren =, <>, <, <=, >, >= sind wie im Falle von Intervallen mengentheoretisch definiert und wirken auf alle Komponenten eines Intervallvektors oder einer Intervallmatrix. Für a, b vom Typ *Livector* und A,B vom Typ *Limatrix* gilt:

$$\begin{array}{lllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[i] & <= & \mathrm{b}[i] \quad \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[i,j] & <= & \mathrm{B}[i,j] \quad \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *Linterval*.

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in“ zwischen einem *Lrvector*- und einem *Livector*-Operanden bzw. zwischen einem *Lrmatrix*- und einem *Limatrix*-Operanden und für die Relation „enthalten im Innern“ zwischen zwei *Livector*-Operanden bzw. *Limatrix*-Operanden sowie >< für den Test auf Disjunktheit zweier Intervallvektoren bzw. Intervallmatrizen zur Verfügung. Diese Operatoren sind jeweils komponentenweise definiert. Die Verbandsoperatoren +* und ** bezeichnen die komponentenweise Bildung der Intervall-Hülle bzw. des Durchschnitts, wie es bereits für den Typ *interval* auf Seite 76 beschrieben wurde.

7.5.3 Transferfunktionen

Zur Wandlung zwischen den Typen *Lrvector* und *Livector* bzw. *Livector* und *ivector* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
intval (lv1,lv2)	*Livector*	Intervallvektor *liv* mit $liv[i] = \text{lintval}\,(lv1[i], lv2[i])$
intval (lv)	*Livector*	Punktintervallvektor *liv* mit $liv[i] = \text{lintval}\,(lv[i])$
inf (liv)	*Lrvector*	Vektor *lv* der Untergrenzen mit $lv[i] = \inf\,(liv[i])$
sup (liv)	*Lrvector*	Vektor *lv* der Obergrenzen mit $lv[i] = \sup\,(liv[i])$
short (liv)	*ivector*	$iv := \text{short}(liv);\ \ liv[i] \subset iv[i]$

lv, lv1, lv2 = *Lrvector*-Ausdruck mit: lv1 <= lv2; liv = *Livector*-Ausdruck
iv = *ivector-Variable*

Zur Wandlung zwischen den Typen *Lrmatrix* und *Limatrix* bzw. *Limatrix* und *imatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergebnistyp**	**Bedeutung**
intval (lM1,lM2)	*Limatrix*	Intervallmatrix liM mit $liM[i,j] =$ lintval $(lM1[i,j], lM2[i,j])$
intval (lM)	*Limatrix*	Punktintervallmatrix liM mit $liM[i,j] =$ lintval $(lM[i,j])$
inf (liM)	*Lrmatrix*	Matrix lM der Untergrenzen mit $lM[i,j] = \inf\ (liM[i,j])$
sup (liM)	*Lrmatrix*	Matrix lM der Obergrenzen mit $lM[i,j] = \sup\ (liM[i,j])$
short (liM)	*imatrix*	$iM :=$ short(liM); $liM[i,j] \subset iM[i,j]$

lM, lM1, lM2 = *Lrmatrix*-Ausdruck mit: lM1 <= lM2; liM = *Limatrix*-Ausdruck
liM = *Limatrix*-Ausdruck; iM = *imatrix*-Variable

7.5.4 Überladungen des Zuweisungsoperators

Zuweisung			**Bedeutung**			
liv	:=	lr	liv [j]	:=	lintval (lr) ;	j = lb(liv),... ,ub(liv)
liv	:=	li	liv [j]	:=	li ;	j = lb(liv),... ,ub(liv)
liv	:=	lv	liv	:=	intval (lv) ;	
liM	:=	lr	liM [j,k]	:=	lintval (lr) ;	j = lb(liM,1),... ,ub(liM,1) k = lb(liM,2),... ,ub(liM,2)
liM	:=	li	liM [j,k]	:=	li ;	j = lb(liM,1),... ,ub(liM,1) k = lb(liM,2),... ,ub(liM,2)
liM	:=	lM	liM	:=	intval (lM) ;	
liM	:=	iM	liM [j,k]	:=	iM [j,k] ;	j = lb(liM,1),... ,ub(liM,1) k = lb(liM,2),... ,ub(liM,2)
liv	:=	iv	liv [j]	:=	iv[j] ;	j = lb(liv),... ,ub(liv)

Tabelle 7.12: Überladungen des Zuweisungsoperators

li = *Linterval*-Ausdruck, liv = *Livector*-**Variable**, liM = *Limatrix*-**Variable**
lr = *Longreal*-Ausdruck, lv = *Lrvector*-Ausdruck, lM = *Lrmatrix*-Ausdruck
iM = *imatrix*-Ausdruck, iv = *ivector*-**Variable**

7.5.5 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktionen *mid* und *diam* für die komponentenweise Berechnung von Mittelpunkt und Durchmesser sowie die Funktion *transp* zur Berechnung der transponierten Intervallmatrix zur Verfügung. Darüberhinaus werden die komponentenweise definierten Funktionen *blow* und *blow_z* für die Epsilonaufblähung bereitgestellt.

Funktion	Ergebnistyp	Bedeutung
null (liv)	*Lrvector*	Nullvektor mit dem aktuellen Indexbereich von liv
null (liM)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen von liM
null (liM1,liM2)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $liM1 \cdot liM2$
id (liM)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von liM
id (liM1,liM2)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $liM1 \cdot liM2$
mid (liv)	*Lrvector*	Mittelpunktsvektor pv mit $pv\,[\,i\,] = mid\,(liv\,[\,i\,])$
diam (liv)	*Lrvector*	Durchmesservektor dv mit $dv\,[\,i\,] = diam\,(liv\,[\,i\,])$
mid (liM)	*Lrmatrix*	Mittelpunktsmatrix pM mit $pM\,[\,i,j\,] = mid\,(liM\,[\,i,j\,])$
diam (liM)	*Lrmatrix*	Durchmessermatrix dM mit $dM\,[\,i,j\,] = diam\,(liM\,[\,i,j\,])$
transp (liM)	*Limatrix*	Transponierte Matrix $liMt$ von liM mit $liMt\,[\,i,j\,] = liM\,[\,j,i\,]$
blow (liv,r)	*Livector*	Vektorielle Epsilonaufblähung ev mit $ev\,[\,i\,] = blow\,(liv\,[\,i\,], r)$ $[0,0] \to [-10^{-531}, +10^{-531}]$
blow_z (liv,r)	*Livector*	Wie $blow(liv,r); [0,0] \to [-10^{-255}, +10^{-255}]$
blow (liM,r)	*Limatrix*	Matrix-Epsilonaufblähung eM mit $eM\,[\,i,j\,] = blow\,(liM\,[\,i,j\,], r)$ $[0,0] \to [-10^{-531}, +10^{-531}]$
blow_z (liM,r)	*Limatrix*	Wie $blow(liM,r); [0,0] \to [-10^{-255}, 10^{-255}]$

Tabelle 7.13: Standardfunktionen des Moduls LMVI_ARI

r = *real*-Ausdruck, liv = *Livector*-Ausdruck,
liM, liM1, liM2 = *Limatrix*-Ausdruck

7.5.6 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : Livector);
procedure read (**var** f : text; **var** A : Limatrix);
procedure write (**var** f : text; a : Livector);
procedure write (**var** f : text; A : Limatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines Intervallvektors bzw. einer Intervallmatrix erfolgt komponentenweise entsprechend der Eingabe von *Linterval*-Werten, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines Intervallvektors bzw. einer Intervallmatrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *Longreal*-Größen aus Tabelle 3.4.

7.5.7 Exakte Auswertung von Ausdrücken

Mit Hilfe des Moduls **MVI_ARI** lassen sich *ivector*-und *imatrix*-Ausdrücke **maximalgenau** auswerten. Bedeutet z.B. A vom Typ *imatrix* die quadratische Koeffizientenmatrix eines linearen Gleichungssystems $A \cdot x = b$, R vom Typ *imatrix* die Näherungsinverse $R \approx A^{-1}$ und ist $id(A)$ die Einheitsmatrix mit dem Indexbereich von A, so liefert die Anweisung

$$D := \#\# \, (id(A) - R * A)$$

die Defektmatrix D vom Typ *imatrix*, wobei bzgl. jeder Matrixkomponente nur eine einzige Rundung pro Intervallgrenze zur nächstkleineren bzw. nächstgrößeren Rasterzahl erfolgt.

Da der obige ##-Ausdruck mit A vom Typ *Limatrix* nicht anwendbar ist, soll die maximalgenaue Auswertung von *Livector*- und *Limatrix*-Ausdrücken simuliert werden! Aus Speicherplatzgründen lassen sich aber jetzt z.B. keine Operatoren für *Limatrix*-Operanden definieren, wenn für jede Ergebniskomponente eine *idotprecision*-Variable erforderlich wird. Man muß vielmehr zur Auswertung eines *Limatrix*-Ausdrucks jede Ergebniskomponente **einzeln** in einer *idotprecision*-Variablen rundungsfehlerfrei berechnen und ihren Wert dann anschließend mit Hilfe der Funktion LONG (vergl. Seite 89) mit nur einer einzigen Rundung pro Intervallgrenze durch eine *Linterval*-Variable einschließen. Dadurch benötigt man für die Auswertung eines beliebigen *Limatrix*-Ausdrucks nur eine einzige *idotprecision*-Variable; man muß jedoch die Ausdruckauswertung in einer Laufanweisung programmieren. Neben den Operatoren *plus_i, minus_i, times_i* aus dem Modul LI_ARI benötigt man dazu aus dem Modul LMVI_ARI lediglich die sieben zusätzlichen Operatoren *times_i* aus Tabelle 7.11 mit *rvector/Lrvector/ivector/Livector*-Operanden und dem Ergebnistyp *idotprecision*. Mit diesen Operatoren lassen sich dann auch *Limatrix*-Ausdrücke maximalgenau auswerten, die zusätzlich noch *imatrix*- oder *rmatrix*-Operanden enthalten. Für die maximalgenaue Auswertung eines *Livector*-Ausdrucks gelten ganz entsprechende Aussagen.

Die beiden folgenden Programmbeispiele zeigen die typischen Vorgehensweisen bei der maximalgenauen Auswertung eines *Limatrix*-Ausdrucks:

```
program Beispiel1 (input, output);

{*  Defekteinschließung eines linearen Gleichungssystems    A · x = b         *}
{*  Ist R ≈ A^-1 eine Näherungsinverse, so liefert die Matrix D: Limatrix    *}
{*  die maximalgenaue Einschließung des Defekts    E − R · A   , wobei        *}
{*  E die Einheitsmatrix mit dem Indexbereich von A ist.                       *}

use lmvi_ari, lmv_ari, li_ari;
var A, D, R                    : Limatrix [1..3, 1..3];
    E                          : Lrmatrix [1..3, 1..3];
    i, k                       : integer;
    idot                       : idotprecision;

begin
   E := id (A);    { E = Einheitsmatrix mit Indexbereich von A }
   For i := lb(E,1) to ub(E,1) do
     For k := lb(E,2) to ub(E,2) do
     begin
       idot := E [ i, k ] minus_i R [ i ] times_i LIVECTOR( A [ *, k ] );
       D [ i, k ] := LONG( idot );
     end;
end.
```

Anmerkungen:

1. In diesem Beispiel wird angenommen, daß die Koeffizienten der Matrix A **Intervalle** sind, so daß *A,R* vom Typ *Limatrix* zu wählen sind.

2. Die Konstruktion LIVECTOR (A $[\,*, k\,]$) liefert dem Compiler die Information, das Teilfeld A $[\,*, k\,]$ als Vektor vom Typ *Livector* zu behandeln.

3. Um die Matrix D maximalgenau berechnen zu können, müssen alle Operatoren in der Anweisung für die *idotprecision*-Variable idot vom Ergebnistyp *dotprecision/idotprecision* sein!

4. Der Operator times_i mit dem Ergebnistyp *idotprecision* und *Livector*-Operanden ist im Modul LMVI_ARI definiert; vergleiche Tabelle 7.11

5. Der Operator minus_i mit *Longreal*- und *idotprecision*- Operanden liefert den Ergebnistyp *idotprecision*; vergl. Tabelle 4.5

Im nächsten Programmbeispiel wird wieder der Defekt $E - R \cdot A$ eines linearen Gleichungssystems $A \cdot x = b$ maximalgenau eingeschlossen, wenn $R \approx A^{-1}$ eine berechnete Näherungsinverse ist. Jetzt wird jedoch angenommen, daß die Koeffizienten der Matrix A Dezimalzahlen sind, d.h. *A,R* müssen vom Typ *Lrmatrix* gewählt werden.

```
program Beispiel2 (input, output);

{* Defekteinschließung eines linearen Gleichungssystems    A · x = b         *}
{* Ist R ≈ A^-1 eine Näherungsinverse, so liefert die Matrix D: Limatrix    *}
{* die maximalgenaue Einschließung des Defekts    E − R · A   , wobei        *}
{* E die Einheitsmatrix mit dem Indexbereich von A ist.                      *}

use lmvi_ari, lmv_ari, li_ari;
var D                        : Limatrix [1..3, 1..3];
    A,E,R                    : Lrmatrix [1..3, 1..3];
    i, k                     : integer;
    idot                     : idotprecision;

begin
   E := lid (A);    { E = Einheitsmatrix mit Indexbereich von A }
   For i := lb(E,1) to ub(E,1) do
     For k := lb(E,2) to ub(E,2) do
     begin
       idot := E [i, k] − R [i] times LVECTOR( A [*, k] );
       D [i, k] := LONG( idot );
     end;
end.
```

Anmerkungen:

1. In diesem Beispiel wird angenommen, daß die Koeffizienten der Matrix A *Longreal*-Zahlen sind, so daß A,R vom Typ *Lrmatrix* zu wählen sind.

2. Die Konstruktion LVECTOR (A $[*, k]$) liefert dem Compiler die Information, das Teilfeld A $[*, k]$ als Vektor vom Typ *Lrvector* zu behandeln.

3. Um die Matrix D maximalgenau berechnen zu können, müssen alle Operatoren in der Anweisung für die *idotprecision*-Variable idot vom Ergebnistyp *dotprecision/idotprecision* sein!

4. Der Operator times mit dem Ergebnistyp *dotprecision* und *Lrvector*-Operanden ist im Modul LMV_ARI definiert; vergleiche Tabelle 7.5

5. Der Operator $-$ mit *Longreal*- und *dotprecision*- Operanden liefert den Ergebnistyp *dotprecision*; vergl. Tabelle 3.9

6. Die Anweisung D $[i, k]$:= LONG(idot) liefert die **bestmögliche**, d.h. maximalgenaue Einschließung des Intervalls *idot* vom Typ *idotprecision* durch ein Intervall D $[i, k]$ vom Typ *Linterval*; vergl. Tabelle 4.7 auf Seite 89.

7.6 Das Modul MVC_ARI

In diesem Modul werden die für das Rechnen mit komplexen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **real** bereitgestellt.

7.6.1 Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Vektoren und Matrizen sind entsprechend der Vereinbarung

type cvector = **dynamic array** [*] **of** complex;
cmatrix = **dynamic array** [*] **of** cvector;

im Sprachkern von PASCAL–XSC enthalten. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.6.2 Operatoren

Viele der bekannten grundlegenden komplexen Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Vektoren und Matrizen komponentenweise erklärt sind gemäß

c	:=	a	±	b	mit	c [i]	:=	a [i] ± b [i]
C	:=	A	±	B	mit	C [i,j]	:=	A [i,j] ± B [i,j]

mit a, b, c vom Typ *cvector* und A, B, C vom Typ *cmatrix*, sind die Operationen $*$ und / definiert durch

s	:=	a	*	b	mit	s	:=	#* (**for** i := lb(a) **to** ub(a) **sum** (a [i] * b [i])) [‡]
c	:=	r	*	a	mit	c [i]	:=	r * a [i]
c	:=	a	*	r	mit	c [i]	:=	a [i] * r
c	:=	a	/	r	mit	c [i]	:=	a [i]/r
c	:=	A	*	b	mit	c [i]	:=	A [i] * b [‡]
C	:=	r	*	A	mit	C [i,j]	:=	r * A [i,j]
C	:=	A	*	r	mit	C [i,j]	:=	A [i,j] * r
C	:=	A	/	r	mit	C [i,j]	:=	A [i,j]/r
C	:=	A	*	B	mit	C [i,j]	:=	A [i] * B [*, j] [‡]

[‡]Maximalgenaues Skalarprodukt mit Ergebnistyp *real*

wobei r, s vom Typ *complex*, a, b, c vom Typ *cvector* und A, B, C vom Typ *cmatrix* sind. Die Operationen mit gerichteten Rundungen sind entsprechend definiert.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ wirken auf alle Komponenten eines Vektors oder einer Matrix. Für a, b vom Typ *cvector* und A,B vom Typ *cmatrix* gilt:

$$\begin{array}{lllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[i] & <= & \mathrm{b}[i] \quad \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[i,j] & <= & \mathrm{B}[i,j] \quad \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *complex*. In nachfolgender Tabelle sind alle Operatoren des Moduls MVC_ARI zusammengestellt:

rechter Operand	integer real	complex	rvector	cvector	rmatrix	cmatrix
linker Operand						
monadisch				$+, -$		$+, -$
integer real				$*, *<, *>$		$*, *<, *>$
complex			$*, *<, *>$	$*, *<, *>$	$*, *<, *>$	$*, *<, *>$
rvector		$*, *<, *>$ $/, /<, />$		$\circ$ $\vee$		
cvector	$*, *<, *>$ $/, /<, />$	$*, *<, *>$ $/, /<, />$	$\circ$ $\vee$	$\circ$ $\vee$		
rmatrix		$*, *<, *>$ $/, /<, />$		$*, *<, *>$		$\circ$ $\vee$
cmatrix	$*, *<, *>$ $/, /<, />$	$*, *<, *>$ $/, /<, />$	$*, *<, *>$	$*, *<, *>$	$\circ$ $\vee$	$\circ$ $\vee$

Tabelle 7.14: Operatoren des Moduls MVC_ARI

$$\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>\}$$
$$\vee \in \{=, <>, <, <=, >, >=\}$$

7.6.3 Transferfunktionen

Zur Wandlung zwischen den Typen *rvector* und *cvector* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergb.-Typ**	**Bedeutung**
compl (rv1,rv2)	*cvector*	Komplexer Vektor *cv* mit
		$cv[\,i\,] = compl\ (rv1[\,i\,], rv2[\,i\,])$
compl (rv)	*cvector*	Rein reeller komplexer Vektor *cv* mit
		$cv[\,i\,] = compl\ (rv[\,i\,])$
re (cv)	*rvector*	Realteilvektor *rv*, mit $rv[\,i\,] = re\ (cv[\,i\,])$
im (cv)	*rvector*	Imaginärteilvektor *rv*, mit $rv[\,i\,] = im\ (cv[\,i\,])$

rv, rv1, rv2 = *rvector*-Ausdruck, cv = *cvector*-Ausdruck

Zur Wandlung zwischen den Typen *rmatrix* und *cmatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergb.-Typ**	**Bedeutung**
compl (rM1,rM2)	*cmatrix*	Komplexe Matrix *cM* mit
		$cM[\,i,j\,] = compl\ (rM1[\,i,j\,], rM2[\,i,j\,])$
compl (rM)	*cmatrix*	Rein reelle komplexe Matrix *cM* mit
		$cM[\,i,j\,] = compl\ (rM[\,i,j\,])$
re (cM)	*rmatrix*	Realteilmatrix *rM* mit
		$rM[\,i,j\,] = re\ (cM[\,i,j\,])$
im (cM)	*rmatrix*	Imaginärteilmatrix *rM* mit
		$rM[\,i,j\,] = im\ (cM[\,i,j\,])$

rM, rM1, rM2 = *rmatrix*-Ausdruck, cM = *cmatrix*-Ausdruck

7.6.4 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *cvector*- und *cmatrix*-Variablen sowie die Wandlungen von *rvector* nach *cvector* und *rmatrix* nach *cmatrix* werden in Form überladener Zuweisungen bereitgestellt:

Zuweisung			**Bedeutung**			
cv	:=	r	cv [j]	:=	compl (r)	j = lb(cv),... ,ub(cv)
cv	:=	c	cv [j]	:=	c	j = lb(cv),... ,ub(cv)
cv	:=	rv	cv	:=	compl (rv)	
cM	:=	r	cM [j, k]	:=	compl (r)	j = lb(cM,1),... ,ub(cM,1)
						k = lb(cM,2),... ,ub(cM,2)
cM	:=	c	cM [j, k]	:=	c	j = lb(cM,1),... ,ub(cM,1)
						k = lb(cM,2),... ,ub(cM,2)
cM	:=	rM	cM	:=	compl (rM)	

c = *complex*-Ausdruck, cv = *cvector*-**Variable**
cM = *cmatrix*-**Variable**, r = *real*-Ausdruck
rv = *rvector*-Ausdruck, rM = *rmatrix*-Ausdruck

7.6.5 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktion *conj* für die Konjugation von Vektoren und Matrizen sowie die Funktionen *transp* und *herm* zur Berechnung der transponierten und der hermiteschen Matrix zur Verfügung.

Funktion	**Ergb.-Typ**	**Bedeutung**
null (cv)	*rvector*	Nullvektor mit dem aktuellen Indexbereich von *cv*
null (cM)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen von *cM*
null (cM1,cM2)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $cM1 \cdot cM2$
id (cM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *cM*
id (cM1,cM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $cM1 \cdot cM2$

cv = *cvector*-Ausdruck, cM, cM1, cM2 = *cmatrix*-Ausdruck

Funktion	**Ergb.-Typ**	**Bedeutung**
conj (cv)	*cvector*	Konjugiert komplexer Vektor *cvk* mit $cvk[\,i\,] = conj\ (cv[\,i\,])$
conj (cM)	*cmatrix*	Konjugiert komplexe Matrix *cMk* mit $cMk[\,i,j\,] = conj\ (cM[\,i,j\,])$
transp (cM)	*cmatrix*	Transponierte Matrix cMt von cM mit $cMt\,[\,i,j\,] = cM\,[\,j,i\,]$
herm (cM)	*cmatrix*	Hermitesche Matrix cMh von cM mit $cMh\,[\,i,j\,] = conj\ (cM\,[\,j,i\,])$

cv = *cvector*-Ausdruck, cM = *cmatrix*-Ausdruck

Beispiel:

Für die komplexen Matrizen *M, M1, M2* vom Typ *cmatrix* ergibt, nach Ausführung der Anweisungen

```
M1 := conj( transp (M) );
M2 := herm( M );
```

der logische Ausdruck

```
M1 = M2
```

den Wert *true*.

7.6.6 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : cvector);
procedure read (**var** f : text; **var** A : cmatrix);
procedure write (**var** f : text; a : cvector);
procedure write (**var** f : text; A : cmatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Vektors bzw. einer komplexen Matrix erfolgt komponentenweise entsprechend der Eingabe von *complex*-Größen, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines komplexen Vektors bzw. einer komplexen Matrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *real*-Größen aus Tabelle 3.4.

7.7 Das Modul LMVC_ARI

In diesem Modul werden die für das Rechnen mit komplexen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **real** bereitgestellt.

7.7.1 Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Vektoren und Matrizen sind definiert durch:

type Lcvector = **dynamic array** [∗] **of** Lcomplex;
Lcmatrix = **dynamic array** [∗] **of** Lcvector;

Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.7.2 Operatoren

Viele der bekannten grundlegenden komplexen Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit den drei verschiedenen Rundungsarten zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Vektoren und Matrizen komponentenweise erklärt sind gemäß

$$\begin{array}{lllllllll} \mathrm{c} & := & \mathrm{a} & \pm & \mathrm{b} & \mathrm{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] \pm \mathrm{b}[i] \\ \mathrm{C} & := & \mathrm{A} & \pm & \mathrm{B} & \mathrm{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] \pm \mathrm{B}[i,j] \end{array}$$

mit a,b,c vom Typ *Lcvector* und A, B, C vom Typ *Lcmatrix*, sind die Operationen $*$ und $/$ definiert durch

$$\begin{array}{lllllllll} \mathrm{s} & := & \mathrm{a} & * & \mathrm{b} & \mathrm{mit} & \mathrm{s} & := & \sum_{i=lb(a)}^{ub(a)} a[i] \cdot b[i] \quad ^{\ddagger} \\ \mathrm{c} & := & \mathrm{r} & * & \mathrm{a} & \mathrm{mit} & \mathrm{c}[i] & := & \mathrm{r} * \mathrm{a}[i] \\ \mathrm{c} & := & \mathrm{a} & * & \mathrm{r} & \mathrm{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] * \mathrm{r} \\ \mathrm{c} & := & \mathrm{a} & / & \mathrm{r} & \mathrm{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] / \mathrm{r} \\ \mathrm{c} & := & \mathrm{A} & * & \mathrm{b} & \mathrm{mit} & \mathrm{c}[i] & := & \mathrm{A}[i] * \mathrm{b} \quad ^{\ddagger} \\ \mathrm{C} & := & \mathrm{r} & * & \mathrm{A} & \mathrm{mit} & \mathrm{C}[i,j] & := & \mathrm{r} * \mathrm{A}[i,j] \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{r} & \mathrm{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] * \mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & / & \mathrm{r} & \mathrm{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] / \mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{B} & \mathrm{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i] * \mathrm{B}[*,j] \quad ^{\ddagger} \end{array}$$

‡Maximalgenaues Skalarprodukt mit Rundung zur nächsten *Longreal*-Zahl

wobei r, s vom Typ *Lcomplex*, a, b, c vom Typ *Lcvector* und A, B, C vom Typ *Lcmatrix* sind. Die Operationen mit gerichteten Rundungen sind entsprechend definiert.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ wirken auf alle Komponenten eines Vektors oder einer Matrix. Für a, b vom Typ *Lcvector* und A,B vom Typ *Lcmatrix* gilt:

$$
\begin{array}{lllllll}
\mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[i] & <= & \mathrm{b}[i] \quad \text{für alle } i \\
\mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[i,j] & <= & \mathrm{B}[i,j] \quad \text{für alle } i,j
\end{array}
$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Operanden vom Typ *Lcomplex*. In nachfolgender Tabelle sind alle Operatoren des Moduls LMVC_ARI zusammengestellt:

rechter Operand	Longreal	Lcomplex	Lrvector	Lcvector	Lrmatrix	Lcmatrix
linker Operand						
monadisch				$+, -$		$+, -$
Longreal				$*, *<, *>$		$*, *<, *>$
Lcomplex			$*, *<, *>$	$*, *<, *>$	$*, *<, *>$	$*, *<, *>$
Lrvector		$*, *<, *>$ $/, /<, />$		$\circ$ $\vee$		
Lcvector	$*, *<, *>$ $/, /<, />$	$*, *<, *>$ $/, /<, />$	$\circ$ $\vee$	$\circ$ $\vee$		
Lrmatrix		$*, *<, *>$ $/, /<, />$		$*, *<, *>$		$\circ$ $\vee$
Lcmatrix	$*, *<, *>$ $/, /<, />$	$*, *<, *>$ $/, /<, />$	$*, *<, *>$	$*, *<, *>$	$\circ$ $\vee$	$\circ$ $\vee$

Tabelle 7.15: Operatoren des Moduls LMVC_ARI

$$
\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>\}
$$
$$
\vee \in \{=, <>, <, <=, >, >=\}
$$

7.7.3 Transferfunktionen

Zur Wandlung zwischen den Typen *Lrvector* und *Lcvector* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergb.-Typ	Bedeutung
compl (Lv1,Lv2)	*Lcvector*	Komplexer Vektor *Lcv* mit
		$Lcv[\,i\,] = lcompl\ (Lv1[\,i\,], Lv2[\,i\,])$
compl (Lv)	*Lcvector*	Rein reeller komplexer Vektor *Lcv* mit
		$Lcv[\,i\,] = lcompl\ (Lv[\,i\,])$
re (Lcv)	*Lrvector*	Realteilvektor *Lv*, mit $Lv[\,i\,] = re\ (Lcv[\,i\,])$
im (Lcv)	*Lrvector*	Imag.-teilvektor *Lv*, mit $Lv[\,i\,] = im\ (Lcv[\,i\,])$

Lv, Lv1, Lv2 = *Lrvector*-Ausdruck, Lcv = *Lcvector*-Ausdruck

Zur Wandlung zwischen den Typen *Lrmatrix* und *Lcmatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergb.-Typ	Bedeutung
compl (LM1,LM2)	*Lcmatrix*	Komplexe Matrix *LcM* mit
		$LcM[\,i,j\,] = lcompl\ (LM1[\,i,j\,], LM2[\,i,j\,])$
compl (LM)	*Lcmatrix*	Rein reelle komplexe Matrix *LcM* mit
		$LcM[\,i,j\,] = lcompl\ (LM[\,i,j\,])$
re (LcM)	*Lrmatrix*	Realteilmatrix *LM* mit
		$LM[\,i,j\,] = re\ (LcM[\,i,j\,])$
im (LcM)	*Lrmatrix*	Imaginärteilmatrix *LM* mit
		$LM[\,i,j\,] = im\ (LcM[\,i,j\,])$

LM, LM1, LM2 = *Lrmatrix*-Ausdruck, LcM = *Lcmatrix*-Ausdruck

Während dem Anwender im Modul **LC_ARI** alle gemischten Operationen mit den Datentypen *complex* und *Lcomplex* zur Verfügung stehen, sind im Modul **LMVC_ARI** die gemischten Operationen z.B. mit den Datentypen *cvector* und *Lcvector* nicht definiert.

Mit Hilfe der Funktionen **SHORT, SHORTUP, SHORTDOWN** lassen sich jedoch Vektoren bzw. Matrizen vom Typ *Lcvector* bzw. *Lcmatrix* durch entsprechende Rundungen umwandeln in Vektoren bzw. Matrizen vom Typ *cvector* bzw. *cmatrix*. Mit den entsprechenden ... _TEST–Versionen dieser Funktionen kann überprüft werden, ob durch Rundungen in den denormalisierten Zahlenbereich bei wenigstens einer Komponente hintere Mantissenstellen auf Null gesetzt werden.

Durch Überladungen des Zuweisungsoperators können umgekehrt Vektoren und Matrizen vom Typ *cvector* bzw. *cmatrix* rundungsfehlerfrei umgewandelt werden in Vektoren und Matrizen vom Typ *Lcvector* bzw. *Lcmatrix*.

Funktion	Erg.-Typ	Bedeutung
short (Lcv)	*cvector*	$cv[i] :=$ short $(Lcv[i])$
shortup (Lcv)	*cvector*	$cv[i] :=$ shortup $(Lcv[i])$
shortdown (Lcv)	*cvector*	$cv[i] :=$ shortdown $(Lcv[i])$
short_test (Lcv,err)	*cvector*	$cv[i] :=$ short_test $(Lcv[i], err)$
shortup_test (Lcv,err)	*cvector*	$cv[i] :=$ shortup_test $(Lcv[i], err)$
shortdown_test (Lcv,err)	*cvector*	$cv[i] :=$ shortdown_test $(Lcv[i], err)$
short (LcM)	*cmatrix*	$cM[i] :=$ short $(LcM[i])$
shortup (LcM)	*cmatrix*	$cM[i] :=$ shortup $(LcM[i])$
shortdown (LcM)	*cmatrix*	$cM[i] :=$ shortdown $(LcM[i])$
short_test (LcM,err)	*cmatrix*	$cM[i] :=$ short_test $(LcM[i], err)$
shortup_test (LcM,err)	*cmatrix*	$cM[i] :=$ shortup_test $(LcM[i], err)$
shortdown_test(LcM,err)	*cmatrix*	$cM[i] :=$ shortdown_test$(LcM[i], err)$

Lcv = *Lcvector*-Ausdruck, cv = *cvector*-**Variable**, err = *boolean*-**Variable**
LcM = *Lcmatrix*-Ausdruck; cM = *cmatrix*-**Variable**

7.7.4 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *Lcvector*- und *Lcmatrix*-Variablen sowie die Wandlungen von *Lrvector* nach *Lcvector* und *Lrmatrix* nach *Lcmatrix* werden in Form überladener Zuweisungen bereitgestellt:

Zuweisung	Bedeutung
Lcv := Lr	Lcv $[j]$:= lcompl (Lr) j = lb(Lcv),... ,ub(Lcv)
Lcv := Lc	Lcv $[j]$:= Lc j = lb(Lcv),... ,ub(Lcv)
Lcv := Lv	Lcv := lcompl (Lv)

Lc = *Lcomplex*-Ausdruck, Lcv = *Lcvector*-**Variable**
Lv = *Lrvector*-Ausdruck, Lr = *Longreal*-Ausdruck

Zuweisung			Bedeutung			
LcM	:=	Lr	LcM$[j,k]$	:=	lcompl (Lr)	j = lb(LcM,1),... ,ub(LcM,1) k = lb(LcM,2),... ,ub(LcM,2)
LcM	:=	Lc	LcM$[j,k]$	:=	Lc	j = lb(LcM,1),... ,ub(LcM,1) k = lb(LcM,2),... ,ub(LcM,2)
LcM	:=	LM	LcM	:=	compl (LM)	
Lcv	:=	cv	Lcv$[j]$	=	cv$[j]$	j = lb(Lcv),... ,ub(Lcv)
LcM	:=	cM	LcM$[j,k]$	=	cM$[j,k]$	j = lb(LcM,1),... ,ub(LcM,1) k = lb(LcM,2),... ,ub(LcM,2)

Lc = *Lcomplex*-Ausdruck, Lr = *Longreal*-Ausdruck
cv = *cvector*-Ausdruck, Lcv = *Lcvector*-**Variable**
cM = *cmatrix*-Ausdruck, LcM = *Lcmatrix*-**Variable**,
LM = *Lrmatrix*-Ausdruck

7.7.5 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktion *conj* für die Konjugation von Vektoren und Matrizen sowie die Funktionen *transp* und *herm* zur Berechnung der transponierten und der hermiteschen Matrix zur Verfügung.

Funktion	Ergb.-Typ	Bedeutung
null (Lcv)	*Lrvector*	Nullvektor mit dem aktuellen Indexbereich von *Lcv*
null (LcM)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen von *LcM*
null (LcM1,LcM2)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $LcM1 \cdot LcM2$
id (LcM)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *LcM*
id (LcM1,LcM2)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $LcM1 \cdot LcM2$
conj (Lcv)	*Lcvector*	Konjugiert komplexer Vektor *cvk* mit $cvk[i] = conj\ (Lcv[i])$

Lcv = *Lcvector*-Ausdruck, LcM, LcM1, LcM2 = *Lcmatrix*-Ausdruck

Funktion	Ergb.-Typ	Bedeutung
conj (LcM)	*Lcmatrix*	Konjugiert komplexe Matrix *lcMk* mit $lcMk[i,j] = conj\ (LcM[i,j])$
transp (LcM)	*Lcmatrix*	Transponierte Matrix $lcMt$ von LcM mit $lcMt[i,j] = LcM[j,i]$
herm (LcM)	*Lcmatrix*	Hermitesche Matrix $lcMh$ von LcM mit $lcMh[i,j] = conj\ (LcM[j,i])$

LcM = *Lcmatrix*-Ausdruck

Beispiel:

Für die komplexen Matrizen *M, M1, M2* vom Typ *Lcmatrix* ergibt, nach Ausführung der Anweisungen

M1 := conj(transp (M));
M2 := herm(M);

der logische Ausdruck

M1 = M2

den Wert *true*.

7.7.6 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : Lcvector);
procedure read (**var** f : text; **var** A : Lcmatrix);
procedure write (**var** f : text; a : Lcvector);
procedure write (**var** f : text; A : Lcmatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Vektors bzw. einer komplexen Matrix erfolgt komponentenweise entsprechend der Eingabe von *Lcomplex*-Größen, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines komplexen Vektors bzw. einer komplexen Matrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *Longreal*-Größen aus Tabelle 3.4.

7.7.7 Exakte Auswertung von Ausdrücken

Benutzt man das Modul **MVC_ARI** , so lassen sich *cvector*- und *cmatrix*–Ausdrücke mit Hilfe des Lattenkreuz–Konzepts maximalgenau auswerten. Da jedoch #–Ausdrücke z.B. nicht mit *Lcmatrix*–Operanden verträglich sind, wird hier an einem Programmbeispiel gezeigt, wie man mit den *Lcmatrix*–Operanden A,B,C z.B. den Ausdruck $A + B \cdot C$ ebenfalls maximalgenau berechnen kann:

```
program beispiel (input,output);
use lc_ari, lmvc_ari;
var cdot    : cdotprecision;
    A,B,C,D : Lcmatrix[1..5,1..5];
    i,k     : integer;

begin { Haupt }
  writeln('A = ?'); read(A);        { Die Eingabe einer Matrix  }
  writeln('B = ?'); read(B);        { erfolgt stets zeilenweise }
  writeln('C = ?'); read(C);
  For i := lb(A,1) to ub(A,1) do
    for k := lb(A,2) to ub(A,2) do
      begin
        cdot := A[i,k];
        PADDNACCU( cdot,LCVECTOR(B[i,*]),LCVECTOR(C[*,k]) );
        D[i,k] := LONGNEXT(cdot)
      end;
  writeln(D);                      { Maximalgen. Ergebnis in  D }
end.
```

Anmerkungen:

1. Eine Matrix wird stets zeilenweise eingegeben, wobei jede Zeile durch ein $<$ return $>$ abzuschließen ist.

2. Die Konstruktion LCVECTOR(B$[i,*]$) liefert dem Compiler die Information, den i-ten Zeilenvektor B$[i,*]$ als Vektor vom Typ *Lcvector* zu behandeln.

3. Die Konstruktion LCVECTOR(B$[*,k]$) liefert dem Compiler die Information, den k-ten Spaltenvektor B$[*,k]$ als Vektor vom Typ *Lcvector* zu behandeln.

4. Die Prozedur **PADDNACCU**(..) aus dem Modul **LC_ARI** addiert zur Variablen cdot = A$[i,k]$ das Produkt B$[i,*]\cdot$C$[*,k]$ rundungsfehlerfrei; vergl. Seite 119, Nr. 12.

5. Die Funktion **LONGNEXT**(cdot) rundet den exakten cdot–Wert zur nächsten *Lcomplex*–Zahl D$[i,k]$; vergl. Seite 120.

7.8 Das Modul MVCI_ARI

In diesem Modul werden die für das Rechnen mit komplexen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **real** bereitgestellt.

7.8.1 Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Intervallvektoren und Intervallmatrizen sind entsprechend der Vereinbarung

type civector = **dynamic array** [*] **of** cinterval;
cimatrix = **dynamic array** [*] **of** civector;

im Sprachkern von PASCAL–XSC enthalten. Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.8.2 Operatoren

Viele der bekannten grundlegenden komplexen, intervallmäßigen Matrix/Vektor-Operationen sind in diesem Modul vordefiniert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der komponentenweisen Rundung zum kleinsten einschließenden komplexen Intervall (auch für gemischte Typen) zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Intervallvektoren und Intervallmatrizen komponentenweise erklärt sind gemäß

c	:=	a	±	b	mit	c [i]	:=	a [i] ± b [i]
C	:=	A	±	B	mit	C [i,j]	:=	A [i,j] ± B [i,j]

mit a, b, c vom Typ *civector* und A, B, C vom Typ *cimatrix*, sind die Operationen $*$ und $/$ definiert durch

s	:=	a	*	b	mit	s	:=	## (**for** i := lb(a) **to** ub(a) **sum** (a [i] * b [i])) ‡
c	:=	r	*	a	mit	c [i]	:=	r * a [i]
c	:=	a	*	r	mit	c [i]	:=	a [i] * r
c	:=	a	/	r	mit	c [i]	:=	a [i]/r
c	:=	A	*	b	mit	c [i]	:=	A [i] * b ‡
C	:=	r	*	A	mit	C [i,j]	:=	r * A [i,j]
C	:=	A	*	r	mit	C [i,j]	:=	A [i,j] * r
C	:=	A	/	r	mit	C [i,j]	:=	A [i,j]/r
C	:=	A	*	B	mit	C [i,j]	:=	A [i] * B [*, j] ‡

‡Maximalgenaues Skalarprodukt mit Ergebnistyp *real*

wobei r, s vom Typ *cinterval*, a, b, c vom Typ *civector* und A, B, C vom Typ *cimatrix* sind. Die Operationen mit gemischten Operandentypen sind entsprechend definiert.

Die wie im Falle von Intervallen und komplexen Intervallen mengentheoretisch zu verstehenden Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind auf der Basis von $=$ und $<=$ realisiert, wobei für a, b vom Typ *civector* und A,B vom Typ *cimatrix* gilt:

$$\begin{array}{llllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[i] & <= & \mathrm{b}[i] & \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[i,j] & <= & \mathrm{B}[i,j] & \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Werte vom Typ *cinterval*.

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in“ zwischen einem *rvector*- oder *cvector*- und einem *civector*- Operanden bzw. zwischen einem *rmatrix*- oder *cmatrix*- und einem *cimatrix*-Operanden sowie für die Relation „enthalten im Innern“ zwischen zwei *civector*-Operanden bzw. zwei *cimatrix*-Operanden zur Verfügung. $><$ für den Test auf Disjunktheit zweier komplexer Intervallvektoren bzw. Intervallmatrizen steht ebenfalls zur Verfügung. Diese Operatoren sind jeweils komponentenweise definiert.

Die Verbandsoperatoren $+*$ und $**$ bezeichnen die komponentenweise Bildung der Intervall-Hülle bzw. des Durchschnitts, wie es bereits für den Typ *cinterval* im Modul CI_ARI beschrieben wurde.

Die Übersicht über alle im Modul MVCI_ARI definierten Operatoren wird, bedingt durch die große Zahl von Operatoren, in den beiden nachfolgenden Tabellen gegeben:

rechter Operand	integer real	complex	interval	cinterval	rvector	cvector
linker Operand						
monadisch						
integer real						
complex						
interval						$*$
cinterval					$*$	$*$
rvector				$*, /$		$+*$
cvector			$*, /$	$*, /$	$+*$	$+*$
ivector		$*, /$		$*, /$		$\diamond$, $=, <>$, $+*$
civector	$*, /$	$*, /$	$*, /$	$*, /$	$\diamond$, $=, <>$, $+*$	$\diamond$, $=, <>$, $+*$
rmatrix				$*, /$		
cmatrix			$*, /$	$*, /$		
imatrix		$*, /$		$*, /$		$*$
cimatrix	$*, /$	$*, /$	$*, /$	$*, /$	$*$	$*$

Tabelle 7.16: Operatoren des Moduls MVCI_ARI, Teil 1

$\diamond \in \{+, -, *\} \qquad \vee \in \{=, <>, <, <=, >, >=\}$

rechter Operand	ivector	civector	rmatrix	cmatrix	imatrix	cimatrix
linker Operand						
monadisch		$+, -$				$+, -$
integer real		$*$				$*$
complex	$*$	$*$			$*$	$*$
interval		$*$		$*$		$*$
cinterval	$*$	$*$	$*$	$*$	$*$	$*$
rvector		$\diamond$, $=, <>$,**in** $+*$				
cvector	$\diamond$, $=, <>$,**in** $+*$	$\diamond$, $=, <>$,**in** $+*$				
ivector		$\diamond$, **in**,$\vee, ><$ $+*, **$				
civector	$\diamond$, $\vee, ><$ $+*, **$	$\diamond$, **in**,$\vee, ><$ $+*, **$				
rmatrix		$*$		$+*$		$\diamond$, $=, <>$,**in** $+*$
cmatrix	$*$	$*$	$+*$	$+*$	$\diamond$, $=, <>$,**in** $+*$	$\diamond$, $=, <>$,**in** $+*$
imatrix		$*$		$\diamond$, $=, <>,$ $+*$		$\diamond$, **in**$\vee, ><,$ $+*, **$
cimatrix	$*$	$*$	$\diamond$, $=, <>$ $+*$	$\diamond$, $=, <>,$ $+*$	$\diamond$, $\vee, ><,$ $+*, **$	$\diamond$, **in**$\vee, ><,$ $+*, **$

Tabelle 7.17: Operatoren des Moduls MVCI_ARI, Teil 2

$\diamond \in \{+, -, *\} \qquad \vee \in \{=, <>, <, <=, >, >=\}$

7.8.3 Transferfunktionen für komplexe Intervallvektoren

Zur Wandlung zwischen den Typen *rvector*, *cvector*, *ivector* und *civector* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergb.-Typ	Bedeutung
compl (iv1,iv2)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = compl\ (iv1[\,i\,], iv2[\,i\,])$
compl (rv,iv)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = compl\ (rv[\,i\,], iv[\,i\,])$
compl (iv,rv)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = compl\ (iv[\,i\,], rv[\,i\,])$
compl (iv)	*civector*	Rein reeller komplexer Intervallvektor *civ* mit $civ[\,i\,] = compl\ (iv[\,i\,])$
intval (cv1,cv2)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = intval\ (cv1[\,i\,], cv2[\,i\,])$
intval (rv,cv)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = intval\ (rv[\,i\,], cv[\,i\,])$
intval (cv,rv)	*civector*	Komplexer Intervallvektor *civ* mit $civ[\,i\,] = intval\ (cv[\,i\,], rv[\,i\,])$
intval (cv)	*civector*	Punktintervallvektor *civ* mit $civ[\,i\,] = intval\ (cv[\,i\,])$
re (civ)	*ivector*	Realteilvektor *iv*, mit $iv[\,i\,] = re\ (civ[\,i\,])$
im (civ)	*ivector*	Imaginärteilvektor *iv*, mit $iv[\,i\,] = im\ (civ[\,i\,])$
inf (civ)	*cvector*	Komplexer Vektor *cv* der Untergrenzen mit $cv[i] = inf\,(civ[\,i\,])$
sup (civ)	*cvector*	Komplexer Vektor *cv* der Obergrenzen mit $cv[i] = sup\,(civ[\,i\,])$

rv = *rvector*-Ausdruck, cv, cv1, cv2 = *cvector*-Ausdruck,
iv, iv1, iv2 = *ivector*-Ausdruck, civ = *civector*-Ausdruck

7.8.4 Transferfunktionen für komplexe Intervallmatrizen

Zur Wandlung zwischen den Typen *rmatrix*, *cmatrix*, *imatrix* und *cimatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergb.-Typ	Bedeutung
compl (iM1,iM2)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = compl\ (iM1[i,j], iM2[i,j])$
compl (rM,iM)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = compl\ (rM[i,j], iM[i,j])$
compl (iM,rM)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = compl\ (iM[i,j], rM[i,j])$
compl (iM)	*cimatrix*	Rein reelle komplexe Intervallmatrix *ciM* mit $ciM[i,j] = compl\ (iM[i,j])$
intval (cM1,cM2)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = intval\ (cM1[i,j], cM2[i,j])$
intval (rM,cM)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = intval\ (rM[i,j], cM[i,j])$
intval (cM,rM)	*cimatrix*	Komplexe Intervallmatrix *ciM* mit $ciM[i,j] = intval\ (cM[i,j], rM[i,j])$
intval (cM)	*cimatrix*	Punktintervallmatrix *ciM* mit $ciM[i,j] = intval\ (cM[i,j])$
re (ciM)	*imatrix*	Realteilmatrix *iM* mit $iM[i,j] = re\ (ciM[i,j])$
im (ciM)	*imatrix*	Imaginärteilmatrix *iM* mit $iM[i,j] = im\ (ciM[i,j])$
inf (ciM)	*cmatrix*	Komplexe Matrix *cM* der Untergrenzen mit $cM[i,j] = inf\ (ciM[i,j])$
sup (ciM)	*cmatrix*	Komplexe Matrix *cM* der Obergrenzen mit $cM[i,j] = sup\ (ciM[i,j])$

rM = *rmatrix*-Ausdruck, cM, cM1, cM2 = *cmatrix*-Ausdruck
iM, iM1, iM2 = *imatrix*-Ausdruck, ciM = *cimatrix*-Ausdruck

7.8.5 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *civector*- und *cimatrix*-Variablen sowie die Wandlungen von *rvector* bzw. *cvector* bzw. *ivector* nach *civector* und *rmatrix* bzw. *cmatrix* bzw. *imatrix* nach *cimatrix* werden in Form überladener Zuweisungen bereitgestellt:

Zuweisung			Bedeutung			
civ	:=	r	civ [j]	:=	compl (intval (r))	j = lb(civ),... ,ub(civ)
civ	:=	c	civ [j]	:=	intval (c)	j = lb(civ),... ,ub(civ)
civ	:=	i	civ [j]	:=	compl (i)	j = lb(civ),... ,ub(civ)
civ	:=	ci	civ [j]	:=	ci	j = lb(civ),... ,ub(civ)
civ	:=	rv	civ	:=	compl (intval(rv))	
civ	:=	cv	civ	:=	intval (cv)	
civ	:=	iv	civ	:=	compl (iv)	
ciM	:=	r	ciM [j, k]	:=	compl (intval (r))	j = lb(ciM,1),... ,ub(ciM,1)
						k = lb(ciM,2),... ,ub(ciM,2)
ciM	:=	c	ciM [j, k]	:=	intval (c)	j = lb(ciM,1),... ,ub(ciM,1)
						k = lb(ciM,2),... ,ub(ciM,2)
ciM	:=	i	ciM [j, k]	:=	compl (i)	j = lb(ciM,1),... ,ub(ciM,1)
						k = lb(ciM,2),... ,ub(ciM,2)
ciM	:=	ci	ciM [j, k]	:=	ci	j = lb(ciM,1),... ,ub(ciM,1)
						k = lb(ciM,2),... ,ub(ciM,2)
ciM	:=	rM	ciM	:=	compl (intval (rM))	
ciM	:=	cM	ciM	:=	intval (cM)	
ciM	:=	iM	ciM	:=	compl (iM)	

ci = *cinterval*-**Ausdruck**, civ = *civector*-**Variable**, ciM = *cimatrix*-**Variable**
i = *interval*-Ausdruck, iv = *ivector*-Ausdruck, iM = *imatrix*-Ausdruck
c = *complex*-Ausdruck, cv = *cvector*-Ausdruck, cM = *cmatrix*-Ausdruck
r = *real*-Ausdruck, rv = *rvector*-Ausdruck, rM = *rmatrix*-Ausdruck

7.8.6 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktionen *mid* und *diam* für die komponentenweise Berechnung von Mittelpunkt und Durchmesser, die Funktion *conj* für die Konjugation von komplexen Intervallvektoren und Intervallmatrizen, sowie die Funktionen *transp* und *herm* zur Berechnung der transponierten und der hermiteschen Matrix zur Verfügung. Darüber hinaus wird die Funktion *blow* für die komponentenweise Berechnung der Epsilonaufblähung bereitgestellt.

Funktion	**Ergb.-Typ**	**Bedeutung**
null (civ)	*rvector*	Nullvektor mit dem aktuellen Indexbereich von *civ*
null (ciM)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen von *ciM*
null (ciM1,ciM2)	*rmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $ciM1 \cdot ciM2$
id (ciM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *ciM*
id (ciM1,ciM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $ciM1 \cdot ciM2$
mid (civ)	*cvector*	Mittelpunktsvektor *pv* mit $pv[\,i\,] = mid\ (civ[\,i\,])$
diam (civ)	*rvector*	Durchmesservektor *dv* mit $dv[\,i\,] = diam\ (civ[\,i\,])$
mid (ciM)	*cmatrix*	Mittelpunktsmatrix *pM* mit $pM[\,i,j\,] = mid\ (ciM[\,i,j\,])$
diam (ciM)	*rmatrix*	Durchmessermatrix *dM* mit $dM[\,i,j\,] = diam\ (ciM[\,i\,])$
conj (civ)	*civector*	Konjugiert komplexer Intervallvektor *civk* mit $civk[\,i\,] = conj\ (civ[\,i\,])$
conj (ciM)	*cimatrix*	Konjugiert komplexe Intervallmatrix *ciMk* mit $ciMk[\,i,j\,] = conj\ (ciM[\,i,j\,])$
transp (ciM)	*cimatrix*	Transponierte Matrix $ciMt$ von ciM mit $ciMt\,[\,i,j\,] = ciM\,[\,j,i\,]$
herm (ciM)	*cimatrix*	Hermitesche Matrix $ciMh$ von ciM mit $ciMh\,[\,i,j\,] = conj\ (ciM\,[\,j,i\,])$
blow (civ,r)	*civector*	Vektorielle Epsilonaufblähung *ev* mit $ev\,[\,i\,] = blow\ (civ[\,i\,], r)$
blow (ciM,r)	*cimatrix*	Vektorielle Epsilonaufblähung *eM* mit $eM\,[\,i,j\,] = blow\ (ciM[\,i,j\,], r)$

civ = *civector*-Ausdruck; ciM, ciM1, ciM2 = *cimatrix*-Ausdruck, r = *real*-Ausdruck

Beispiel:

Für die komplexen Matrizen *M, M1, M2* vom Typ *cimatrix* ergibt, nach Ausführung der Anweisungen

M1 := conj(transp (M));
M2 := herm(M);

der logische Ausdruck

M1 = M2

den Wert *true*.

7.8.7 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : civector);
procedure read (**var** f : text; **var** A : cimatrix);
procedure write (**var** f : text; a : civector);
procedure write (**var** f : text; A : cimatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Intervallvektors bzw. einer komplexen Intervallmatrix erfolgt komponentenweise entsprechend der Eingabe von *cinterval*-Größen, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines komplexen Intervallvektors bzw. einer komplexen Intervallmatrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *real*-Größen aus Tabelle 3.4.

7.9 Das Modul LMVCI_ARI

In diesem Modul werden die für das Rechnen mit komplexen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren auf der Grundlage des Datentyps **Longreal** bereitgestellt.

7.9.1 Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Intervallvektoren und Intervallmatrizen sind definiert durch:

type Lcivector = **dynamic array** [$*$] **of** Lcinterval;
Lcimatrix = **dynamic array** [$*$] **of** Lcivector;

Die aktuellen Indexgrenzen werden erst bei der Vereinbarung von Variablen dieser Typen festgelegt.

7.9.2 Operatoren

Viele der bekannten grundlegenden komplexen, intervallmäßigen Matrix/Vektor-Operationen sind in diesem Modul vordefiniert, dabei sind gemischte Operanden mit den Basistypen **real** und **Longreal** nicht realisiert. Als arithmetische Operatoren stehen die monadischen Operatoren $+, -$ und die vier Grundoperationen $+, -, *, /$ mit der komponentenweisen Rundung zum kleinsten einschließenden komplexen Intervall zur Verfügung, wobei jedoch nur bestimmte Operandenkombinationen zulässig sind. Während die Operationen $+$ und $-$ für Intervallvektoren und Intervallmatrizen komponentenweise erklärt sind gemäß

$$\begin{array}{lllllcll} \mathrm{c} & := & \mathrm{a} & \pm & \mathrm{b} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] \pm \mathrm{b}[i] \\ \mathrm{C} & := & \mathrm{A} & \pm & \mathrm{B} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] \pm \mathrm{B}[i,j] \end{array}$$

mit a, b, c vom Typ *Lcivector* und A, B, C vom Typ *Lcimatrix*, sind die Operationen $*$ und / definiert durch

$$\begin{array}{lllllcccl} \mathrm{s} & := & \mathrm{a} & * & \mathrm{b} & \text{mit} & \mathrm{s} & := & \sum_{i=lb(a)}^{ub(a)} a[i] \cdot b[i] \quad ^{\ddagger} \\ \mathrm{c} & := & \mathrm{r} & * & \mathrm{a} & \text{mit} & \mathrm{c}[i] & := & \mathrm{r} * \mathrm{a}[i] \\ \mathrm{c} & := & \mathrm{a} & * & \mathrm{r} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i] * \mathrm{r} \\ \mathrm{c} & := & \mathrm{a} & / & \mathrm{r} & \text{mit} & \mathrm{c}[i] & := & \mathrm{a}[i]/\mathrm{r} \\ \mathrm{c} & := & \mathrm{A} & * & \mathrm{b} & \text{mit} & \mathrm{c}[i] & := & \mathrm{A}[i] * \mathrm{b} \quad ^{\ddagger} \\ \mathrm{C} & := & \mathrm{r} & * & \mathrm{A} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{r} * \mathrm{A}[i,j] \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{r} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j] * \mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & / & \mathrm{r} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i,j]/\mathrm{r} \\ \mathrm{C} & := & \mathrm{A} & * & \mathrm{B} & \text{mit} & \mathrm{C}[i,j] & := & \mathrm{A}[i] * \mathrm{B}[*,j] \quad ^{\ddagger} \end{array}$$

‡Maximalgenaues Skalarprodukt mit Ergebnistyp *Longreal*

wobei r, s vom Typ *Lcinterval*, a, b, c vom Typ *Lcivector* und A, B, C vom Typ *Lcimatrix* sind. Die Operationen mit gemischten Operandentypen sind entsprechend definiert.

Die wie im Falle von Intervallen und komplexen Intervallen mengentheoretisch zu verstehenden Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind auf der Basis von $=$ und $<=$ realisiert, wobei für a, b vom Typ *Lcivector* und A,B vom Typ *Lcimatrix* gilt:

$$\begin{array}{lllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a}[\,i\,] & <= & \mathrm{b}[\,i\,] \quad \text{für alle } i \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A}[\,i,j\,] & <= & \mathrm{B}[\,i,j\,] \quad \text{für alle } i,j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Werte vom Typ *Lcinterval*.

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in" zwischen einem *Lrvector*- oder *Lcvector*- und einem *Lcivector*- Operanden bzw. zwischen einem *Lrmatrix*- oder *Lcmatrix*- und einem *Lcimatrix*-Operanden sowie für die Relation „enthalten im Innern" zwischen zwei *Lcivector*-Operanden bzw. zwei *Lcimatrix*-Operanden zur Verfügung. $><$ für den Test auf Disjunktheit zweier komplexer Intervallvektoren bzw. Intervallmatrizen steht ebenfalls zur Verfügung. Diese Operatoren sind jeweils komponentenweise definiert.

Die Verbandsoperatoren $+*$ und $**$ bezeichnen die komponentenweise Bildung der Intervall-Hülle bzw. des Durchschnitts, wie es bereits für den Typ *Lcinterval* im Modul LCI_ARI beschrieben wurde.

Die Übersicht über alle im Modul LMVCI_ARI definierten Operatoren wird, bedingt durch die große Zahl von Operatoren, in den beiden nachfolgenden Tabellen gegeben:

rechter Operand	Longreal	Lcomplex	Linterval	Lcinterval	Lrvector	Lcvector
linker Operand						
monadisch						
Longreal						
Lcomplex						
Linterval						$*$
Lcinterval					$*$	$*$
Lrvector				$*, /$		$+*$
Lcvector			$*, /$	$*, /$	$+*$	$+*$
Livector		$*, /$		$*, /$		$\diamond$, $=, <>$, $+*$
Lcivector	$*, /$	$*, /$	$*, /$	$*, /$	$\diamond$, $=, <>$, $+*$	$\diamond$, $=, <>$, $+*$
Lrmatrix				$*, /$		
Lcmatrix			$*, /$	$*, /$		
Limatrix		$*, /$		$*, /$		$*$
Lcimatrix	$*, /$	$*, /$	$*, /$	$*, /$	$*$	$*$

Tabelle 7.18: Operatoren des Moduls LMVCI_ARI, Teil 1

$\diamond \in \{+, -, *\}$ $\qquad$ $\vee \in \{=, <>, <, <=, >, >=\}$

rechter Operand	Livector	Lcivector	Lrmatrix	Lcmatrix	Limatrix	Lcimatrix
linker Operand						
monadisch		+, −				+, −
Longreal		*				*
Lcomplex	*	*			*	*
Linterval		*		*		*
Lcinterval	*	*	*	*	*	*
Lrvector		$\diamond$, =, <>,**in** +*				
Lcvector	$\diamond$, =, <>,**in** +*	$\diamond$, =, <>,**in** +*				
Livector		$\diamond$, **in**,$\vee$, >< +*, **				
Lcivector	$\diamond$, $\vee$, >< +*, **	$\diamond$, **in**,$\vee$, >< +*, **				
Lrmatrix		*		+*		$\diamond$, =, <>,**in** +*
Lcmatrix	*	*	+*	+*	$\diamond$, =, <>,**in** +*	$\diamond$, =, <>,**in** +*
Limatrix		*		$\diamond$, =, <>, +*		$\diamond$, **in**$\vee$, ><, +*, **
Lcimatrix	*	*	$\diamond$, =, <> +*	$\diamond$, =, <>, +*	$\diamond$, $\vee$, ><, +*, **	$\diamond$, **in**$\vee$, ><, +*, **

Tabelle 7.19: Operatoren des Moduls LMVCI_ARI, Teil 2

$\diamond \in \{+, -, *\} \qquad \vee \in \{=, <>, <, <=, >, >=\}$

7.9.3 Transferfunktionen für komplexe Intervallvektoren

Zur Wandlung zwischen den Typen *Lrvector*, *Lcvector*, *Livector* und *Lcivector* sowie zwischen *Lcivector* und *civector* werden folgende Transferfunktionen bereitgestellt:

Funktion	**Ergb.-Typ**	**Bedeutung**
compl (liv1,liv2)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lcompl\ (liv1[\,i\,], liv2[\,i\,])$
compl (lv,liv)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lcompl\ (lv[\,i\,], liv[\,i\,])$
compl (liv,lrv)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lcompl\ (liv[\,i\,], lv[\,i\,])$
compl (liv)	*Lcivector*	Rein reeller komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lcompl\ (liv[\,i\,])$
intval (lcv1,lcv2)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lintval\ (lcv1[\,i\,], lcv2[\,i\,])$
intval (lv,lcv)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lintval\ (lv[\,i\,], lcv[\,i\,])$
intval (lcv,lv)	*Lcivector*	Komplexer Intervallvektor *lciv* mit $lciv[\,i\,] = lintval\ (lcv[\,i\,], lv[\,i\,])$
intval (lcv)	*Lcivector*	Punktintervallvektor *lciv* mit $lciv[\,i\,] = lintval\ (lcv[\,i\,])$
short (lciv)	*civector*	lciv$[\,i\,]$ $\subseteq$ short (lciv$[\,i\,]$)
re (lciv)	*Livector*	Realteilvektor *liv*, mit $liv[\,i\,] = re\ (lciv[\,i\,])$
im (lciv)	*Livector*	Imaginärteilvektor *liv*, mit $liv[\,i\,] = im\ (lciv[\,i\,])$
inf (lciv)	*Lcvector*	Komplexer Vektor *lcv* der Untergrenzen mit $lcv[i] = inf\,(lciv[\,i\,])$
sup (lciv)	*Lcvector*	Komplexer Vektor *lcv* der Obergrenzen mit $lcv[i] = sup\,(lciv[\,i\,])$

lv = *Lrvector*-Ausdruck, lcv, lcv1, lcv2 = *Lcvector*-Ausdruck,
liv, liv1, liv2 = *Livector*-Ausdruck, lciv = *Lcivector*-Ausdruck

7.9.4 Transferfunktionen für komplexe Intervallmatrizen

Zur Wandlung zwischen den Typen *Lrmatrix*, *Lcmatrix*, *Limatrix* und *Lcimatrix* sowie zwischen *Lcimatrix* und *cimatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergb.-Typ	Bedeutung
compl (liM1,liM2)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lcompl\ (liM1[i,j], liM2[i,j])$
compl (LM,liM)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lcompl\ (LM[i,j], liM[i,j])$
compl (liM,LM)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lcompl\ (liM[i,j], LM[i,j])$
compl (liM)	*Lcimatrix*	Rein reelle komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lcompl\ (liM[i,j])$
intval (lcM1,lcM2)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lintval\ (lcM1[i,j], lcM2[i,j])$
intval (LM,lcM)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lintval\ (LM[i,j], lcM[i,j])$
intval (lcM,LM)	*Lcimatrix*	Komplexe Intervallmatrix *lciM* mit $lciM[i,j] = lintval\ (lcM[i,j], LM[i,j])$
intval (lcM)	*Lcimatrix*	Punktintervallmatrix *lciM* mit $lciM[i,j] = lintval\ (lcM[i,j])$
short (lciM)	*cimatrix*	lciM$[i,j] \subseteq$ short (lciM$[i,j]$)
re (lciM)	*Limatrix*	Realteilmatrix *liM* mit $liM[i,j] = re\ (lciM[i,j])$
im (lciM)	*Limatrix*	Imaginärteilmatrix *liM* mit $liM[i,j] = im\ (lciM[i,j])$
inf (lciM)	*Lcmatrix*	Komplexe Matrix *lcM* der Untergrenzen mit $lcM[i,j] = inf\ (lciM[i,j])$
sup (lciM)	*Lcmatrix*	Komplexe Matrix *lcM* der Obergrenzen mit $lcM[i,j] = sup\ (lciM[i,j])$

LM = *Lrmatrix*-Ausdruck, lcM, lcM1, lcM2 = *Lcmatrix*-Ausdruck
liM, liM1, liM2 = *Limatrix*-Ausdruck, lciM = *Lcimatrix*-Ausdruck

7.9.5 Überladungen des Zuweisungsoperators

Die komponentenweise Initialisierung von *Lcivector*- und *Lcimatrix*-Variablen sowie die Wandlungen von *Lrvector* bzw. *Lcvector* bzw. *Livector* nach *Lcivector* und *Lrmatrix* bzw. *Lcmatrix* bzw. *Limatrix* nach *Lcimatrix* und *civector* bzw. *cimatrix* nach *Lcivector* bzw. *Lcimatrix* werden in Form überladener Zuweisungen bereitgestellt:

Zuweisung	**Bedeutung**		
lciv := Lr	lciv [j]	:= lcompl (lintval (Lr))	j = lb(lciv),... ,ub(lciv)
lciv := lc	lciv [j]	:= lintval (lc)	j = lb(lciv),... ,ub(lciv)
lciv := li	lciv [j]	:= lcompl (li)	j = lb(lciv),... ,ub(lciv)
lciv := lci	lciv [j]	:= lci	j = lb(lciv),... ,ub(lciv)
lciv := civ	lciv [j]	:= Long($civ[\,j\,]$)	j = lb(lciv),... ,ub(lciv)
lciv := lv	lciv	:= lcompl (lintval(lv))	
lciv := lcv	lciv	:= lintval (lcv)	
lciv := iv	lciv	:= lcompl (liv)	
lciM := Lr	lciM [j,k]	:= lcompl (lintval (Lr))	j = lb(lciM,1),... ,ub(lciM,1) k = lb(lciM,2),... ,ub(lciM,2)
lciM := lc	lciM [j,k]	:= lintval (lc)	j = lb(lciM,1),... ,ub(lciM,1) k = lb(lciM,2),... ,ub(lciM,2)
lciM := li	lciM [j,k]	:= lcompl (li)	j = lb(lciM,1),... ,ub(lciM,1) k = lb(lciM,2),... ,ub(lciM,2)
lciM := lci	lciM [j,k]	:= lci	j = lb(lciM,1),... ,ub(lciM,1) k = lb(lciM,2),... ,ub(lciM,2)
lciM := ciM	lciM [j,k]	:= Long($ciM[\,j,k\,]$)	j = lb(lciM,1),... ,ub(lciM,1) k = lb(lciM,2),... ,ub(lciM,2)
lciM := LM	lciM	:= lcompl (lintval (LM))	
lciM := lcM	lciM	:= lintval (lcM)	
lciM := liM	lciM	:= lcompl (liM)	

lci = *Lcinterval*-**Ausdr.**, lciv = *Lcivector*-**Variable**, lciM = *Lcimatrix*-**Variable**
li = *Linterval*-Ausdruck, liv = *Livector*-Ausdruck, liM = *Limatrix*-Ausdruck
lc = *Lcomplex*-Ausdruck, lcv = *Lcvector*-Ausdruck, lcM = *Lcmatrix*-Ausdruck
Lr = *Longreal*-Ausdruck, lv = *Lrvector*-Ausdruck, LM = *Lrmatrix*-Ausdruck
civ = *civector*-Ausdruck, ciM = *cimatrix*-Ausdruck, iv = *ivector*-Ausdruck

7.9.6 Standardfunktionen

Es stehen die Funktionen *id* und *null* zur Erzeugung einer Einheitsmatrix und einer Nullmatrix bzw. eines Nullvektors, die Funktionen *mid* und *diam* für die komponentenweise Berechnung von Mittelpunkt und Durchmesser, die Funktion *conj* für die Konjugation von komplexen Intervallvektoren und Intervallmatrizen, sowie die Funktionen *transp* und *herm* zur Berechnung der transponierten und der hermiteschen Matrix zur Verfügung. Darüber hinaus wird die Funktion *blow* für die komponentenweise Berechnung der Epsilonaufblähung bereitgestellt.

Funktion	**Ergb.-Typ**	**Bedeutung**
null (lciv)	*Lrvector*	Nullvektor mit dem aktuellen Indexbereich von *lciv*
null (lciM)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen von *lciM*
null (lciM1,lciM2)	*Lrmatrix*	Nullmatrix mit den aktuellen Indexbereichen der Produktmatrix $lciM1 \cdot lciM2$
id (lciM)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *lciM*
id (lciM1,lciM2)	*Lrmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix $lciM1 \cdot lciM2$
mid (lciv)	*Lcvector*	Mittelpunktsvektor *pv* mit $pv[\,i\,] = mid\ (lciv[\,i\,])$
diam (lciv)	*Lrvector*	Durchmesservektor *dv* mit $dv[\,i\,] = diam\ (lciv[\,i\,])$
mid (lciM)	*Lcmatrix*	Mittelpunktsmatrix *pM* mit $pM[\,i,j\,] = mid\ (lciM[\,i,j\,])$
diam (lciM)	*Lrmatrix*	Durchmessermatrix *dM* mit $dM[\,i,j\,] = diam\ (lciM[\,i\,])$
conj (lciv)	*Lcivector*	Konjugiert komplexer Intervallvektor *lcivk* mit $lcivk[\,i\,] = conj\ (lciv[\,i\,])$
conj (lciM)	*Lcimatrix*	Konjugiert komplexe Intervallmatrix *lciMk* mit $lciMk[\,i,j\,] = conj\ (lciM[\,i,j\,])$
transp (lciM)	*Lcimatrix*	Transponierte Matrix $lciMt$ von $lciM$ mit $lciMt\,[\,i,j\,] = lciM\,[\,j,i\,]$
herm (lciM)	*Lcimatrix*	Hermitesche Matrix $lciMh$ von $lciM$ mit $lciMh\,[\,i,j\,] = conj\ (lciM\,[\,j,i\,])$
blow (lciv,Lr)	*Lcivector*	Vektorielle Epsilonaufblähung *ev* mit $ev\,[\,i\,] = blow\ (lciv[\,i\,], short(Lr))$
blow (lciM,Lr)	*Lcimatrix*	Vektorielle Epsilonaufblähung *eM* mit $eM\,[\,i,j\,] = blow\ (lciM[\,i,j\,], short(Lr))$

lciv = *Lcivector*-Ausdruck; lciM, lciM1, lciM2 = *Lcimatrix*-Ausdruck,
Lr = *Longreal*-Ausdruck

Beispiel:

Für die komplexen Matrizen *M, M1, M2* vom Typ *Lcimatrix* ergibt, nach Ausführung der Anweisungen

M1 := conj(transp (M));
M2 := herm(M);

der logische Ausdruck

M1 = M2

den Wert *true*.

7.9.7 Ein-/Ausgabeprozeduren

Es stehen die Prozeduren

procedure read (**var** f : text; **var** a : Lcivector);
procedure read (**var** f : text; **var** A : Lcimatrix);
procedure write (**var** f : text; a : Lcivector);
procedure write (**var** f : text; A : Lcimatrix);

mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

Die Eingabe eines komplexen Intervallvektors bzw. einer komplexen Intervallmatrix erfolgt komponentenweise entsprechend der Eingabe von *lcinterval*-Größen, dabei wird eine Matrix stets zeilenweise eingelesen. Die Ausgabe eines komplexen Intervallvektors bzw. einer komplexen Intervallmatrix erfolgt wie bei der Eingabe komponentenweise mit dem Standardformat für *Longreal*-Größen aus Tabelle 3.4.

Die Eingabe einer dreidimensionalen, komplexen Intervallmatrix

$$\begin{pmatrix} 1+i & 2+i\cdot[1,3] & [1.4,1.5]+i\cdot 0.1 \\ \mathrm{[1,2]+i} & i\cdot[-1,1] & i \\ [-2.3,1.22]+i\cdot[-3,-2.99] & 0 & 1.25\mathrm{E}{-013} \end{pmatrix}$$

kann erfolgen in der Form:

$$\begin{array}{cccc} (1,1) & (2,[1,3]) & ([1.4,1.5],0.1) & <\mathbf{return}> \\ ([1,2],1) & (0,[-1,1]) & (0,1) & <\mathbf{return}> \\ ([-2.3,1.22],[-3,-2.99]) & 0 & 1.25\mathrm{E}{-013} & <\mathbf{return}> \end{array}$$

$(1,1)$ bedeutet hier also ein komplexes Punktintervall und $(0,[-1,1])$ steht für ein rein imaginäres Intervall.

7.9.8 Exakte Auswertung von Ausdrücken

Zu berechnen ist das komplexe Intervall-Skalarprodukt

$$(A \cdot b) \cdot c$$

das im folgenden Programm zunächst **exakt** ausgewertet und anschließend in das nächstgrößere Intervall vom Typ *Lcinterval* gerundet wird:

```
program Beispiel (input,output);
use lci_ari, lmvci_ari;
var cidot,
    cidot_su : cidotprecision;
    A        : Lcimatrix[1..3,1..3];{ Kompl. Intervall-Matrix   }
    b,c      : Lcivector[1..3];     { Kompl. Intervall-Vektoren }
    sk_pr    : Lcinterval;          { Skalar-Produkt-Einschlg.  }
    i        : integer;

begin
  writeln('A: Lcimatrix = ?'); read(A);  { Eingabe: zeilenweise }
  writeln('b: Lcivector = ?'); read(b);
  writeln('c: Lcivector = ?'); read(c);
  cidot_su := 0;                  { Initialisierung }
  For i := lb(a) to ub(a) do
  begin
    cidot := 0;                   { Initialisierung }
    PADDNACCU( cidot,LCIVECTOR(A[i,*]),b ); { A * b }
    cidot_su := cidot_su + cidot * c[i];   { Skalar-}
  end;                       { Produkt aufsummieren }
  sk_pr := LONG(cidot_su);   { Rundg. nach sk_pr    }
  writeln(sk_pr);
end.
```

Anmerkungen:

1. Das obige Skalarprodukt läßt sich für A: cimatrix; b,c: civector nicht mit dem **PASCAL-XSC** Lattenkreuzkonzept für cimatrix-, civector–Ausdrücke behandeln, es muß ebenfalls in einer analogen Schleife berechnet werden!

2. Die Konstruktion LCIVECTOR(A$[i,*]$) liefert dem Compiler die Information, den i-ten Zeilenvektor A$[i,*]$ als Vektor vom Typ *Lcivector* zu behandeln.

3. Die Prozedur **PADDNACCU**(..) aus dem Modul **LCI_ARI** addiert zur Variablen cidot $= 0$ das Skalarprodukt A$[i,*]\cdot$b rundungsfehlerfrei; vergl. Seite 150, Nr. 17.

4. Die Funktion **LONG**(cidot_su) liefert mit dem komplexen Intervall sk_pr die bestmögliche Einschließung des exakten, komplexe Intervall–Skalarprodukts cidot_su; vergl. Tabelle 6.12 auf Seite 145.

Kapitel 8

PASCAL-XSC Module

Wie jede moderne Programmiersprache besitzt auch diese **PASCAL-XSC** BCD–Version ein Modulkonzept, das dem der **IEEE**–Version von **PASCAL-XSC** genau entspricht. Weitere Einzelheiten zum Gebrauch und zur Entwicklung von Modulen findet man in [5, 6, 7].
Wir unterscheiden zwischen **System-, Hilfs-** und **Anwendermodulen**:

- **Systemmodule** sind: STDMOD, TIMER, ... und bestimmen das zur Verfügung stehende **PASCAL-XSC** BCD–System.

- **Hilfsmodule** sind: DDF_ ARI, LSYS, ... zur Behandlung numerischer Spezialgebiete wie Automatische Differentiation, Lineare Gleichungssysteme, ... Da diese Module erfahrungsgemäß immer wieder erweitert und verbessert werden, stehen entsprechende Updates zur Verfügung. Alle Problemlösungen der Toolbox-Bände I,II [3, 4] sollen so auch für die BCD–Version durch die **Hilfsmodule** bereitgestellt werden. In den entsprechenden Beschreibungen findet der Anwender weniger den schon in den Toolboxbänden behandelten theoretischen Hintergrund sondern schwerpunktmäßig viele Beispiele und Hinweise zu den Lösungsroutinen, womit eine schnelle und problemlose Einarbeitung ermöglicht werden soll.

- **Anwendermodule** enthalten vom Anwender erstelle Problemlösungen, mit dem das **PASCAL-XSC** System nach Bedarf beliebig erweitert werden kann.

8.1 Systemmodule

8.1.1 Modulhierarchie

Durch die **use**-Klausel wird in jedem Modul festgelegt, welche anderen Module zu importieren sind. Dabei spielt das Basis-Modul **STDMOD** ein Ausnahmerolle, weil es automatisch von jedem Modul oder Hauptprogramm benutzt wird und daher nicht in die **use**–Klausel aufgenommen werden muß.

Da dem Anwender mit der BCD–Version die verschiedenen Zahlenformate

integer, real, Longreal, rrG, lrG, dotprecision

zu Verfügung stehen, existiert eine Vielzahl von Modulen, deren Hierarchie nicht geschlossen auf einer Buchseite dargestellt werden kann. In nachfolgender Tabelle sind daher in der ersten Spalte alle Systemmodule in der Reihenfolge angegeben, wie sie vom Compiler zur Übersetzung benötigt werden. In jeder Zeile findet man in den nachfolgenden Spalten alle diejenigen Module, die vom Modul aus der ersten Spalte durch die **use**–Klausel importiert werden:

Modul	**Importierte Module**				
STDMOD					
TIMER					
X_INTG					
X_REAL					
X_STRG					
IOSTD					
LI_ARI					
I_ARI	li_ari				
RRGI_ARI	li_ari				
C_ARIAUX	x_real				
LC_HELP	li_ari				
LC_ARI	lc_help	li_ari			
LCI_ARI	i_ari	li_ari	lc_ari	rrGi_ari	
C_ARI	c_ariaux	i_ari	lc_ari		
CI_ARI	c_ari	i_ari	lci_ari	li_ari	
MV_ARI	iostd				
MVI_ARI	i_ari	mv_ari			
MVC_ARI	c_ari	mv_ari			
MVCI_ARI	ci_ari	i_ari	mvc_ari	mvi_ari	mv_ari
LMV_ARI	iostd				
LMVI_ARI	li_ari	lmv_ari	mv_ari		
LMVC_ARI	lc_ari	lmv_ari			
LMVCI_ARI	lci_ari	li_ari	lmvc_ari	lmvi_ari	lmv_ari
	mvci_ari				

8.1.2 Modulbeschreibungen

In diesem Abschnitt erfolgt nur eine Grobbeschreibung der System-Module. Falls deren Inhalte schon in den vorangegangenen Kapiteln behandelt wurden, findet man lediglich entsprechende Hinweise mit den zugehörigen Seitenzahlen.

8.1.2.1 Basismodul STDMOD

Wie schon anfangs erwähnt, wird das Basismodul **STDMOD** beim Übersetzen eines Moduls oder eines Hauptprogramms automatisch importiert, da es alle im Kapitel 3 beschriebenen Typen, Funktionen, Prozeduren und Operatoren exportiert. Damit werden schon alle reellen Rechnungen mit Punktargumenten vom Typ

integer, real, Longreal, rrG, lrG, dotprecision

ermöglicht. Zusätzlich werden auch alle numerischen Datentypen bereitgestellt, die für Intervallrechnungen, komplexe Rechnungen und komplexe Intervallrechnungen mit Punkt-, Vektor- oder Matrixoperanden notwendig sind:

Exportierte numerische Datentypen					
integer	Longreal	rrGinterval	Linterval	Lrmatrix	dotprecision
real	interval	cmatrix	Lrvector	Lcmatrix	idotprecision
rrG	cinterval	cimatrix	Livector	Lcinterval	cdotprecision
lrG	rvector	imatrix	Lcvector	Limatrix	cidotprecision
complex	cvector	rmatrix	Lcivector	Lcimatrix	rrGcinterval
ivector	civector			Lcomplex	rrGcomplex

Darüber hinaus stehen neben den im Kapitel 3 beschriebenen Hilfsfunktionen und Prozeduren auch alle Standardfunktionen aus den entsprechenden Tabellen auf den Seiten 16,27,58,59,60,61,73 zur Verfügung.

8.1.2.2 TIMER

Das Modul exportiert die folgenden Funktionen und Prozeduren:

1. procedure **init_timer**; Speichert die momentane CPU–Zeit.
2. function **get_timer** : integer; Bestimmt die Laufzeitdifferenz zum letzten **init_timer**–Aufruf in der Einheit Sekunden.
3. function **get_date** : string; Liefert Datum und Uhrzeit in der Form: Wochentag, Monat, Tag–Nr., Uhrzeit, Jahr;
4. function **get_date** (format: string) : string; Liefert Datum- oder Uhrzeit-informationen in Abhängigkeit einer Eingangs–Variablen vom Typ *string*.

Die string–Variable format der Funktion **get_date** bewirkt die in folgender Tabelle zusammengestellten Ausgabeformate:

Bedeutung der string-Variablen format in get_date	
string–Variable	**Ausgabeformat**
format := ' '	Wochentag Monat Monatstag–Nr. Uhrzeit Jahr
format := ' % a '	Abgekürzter Wochentag
format := ' % A '	Ausgeschriebener Wochentag
format := ' % b '	Ausgeschriebener Monat
format := ' % B '	Abgekürzter Monat
format := ' % c '	Datum und Uhrzeit
format := ' % d '	Nur Monatstag–Nr.
format := ' % H '	Stundenangabe im Bereich: 0 – 23
format := ' % I '	Stundenangabe im Bereich: 0 – 12
format := ' % j '	Nur die Nummer des Jahrestages
format := ' % m '	Nur der Monat
format := ' % M '	Nur die Minute der Uhrzeit
format := ' % p '	Nur AM / PM
format := ' % S '	Nur die Sekundenangabe der Uhrzeit
format := ' % w '	Nur die Wochentags–Nr.
format := ' % x '	Datum in Zifferform, z.B. 17/02/39
format := ' % X '	Uhrzeit in Zifferform, z.B. 21:13:33
format := ' % y '	Jahresangabe ohne Jahrhundert, z.B. 97
format := ' % Y '	Jahresangabe mit Jahrhundert, z.B. 1997
format := ' % % '	Ausgabe des % Zeichens

8.1.2.3 X_REAL

Das Modul **x_real** liefert zusätzliche Konstanten, Funktionen und Prozeduren bzgl. der Datentypen **REAL, LONGREAL**. In [7] findet man weitere Informationen zu diesen Modul–Routinen, die für die BCD–Version jedoch nur dann gültig sind, wenn in diesem Abschnitt ausdrücklich auf [7] verwiesen wird. Beachten Sie bitte, daß sich das genannte Handbuch nur auf die **IEEE**–Version bezieht!

8.1.2.3.1 Konstanten

Name	Wert	Bedeutung
IEEE_INV_OP	256	Illegale Operation
IEEE_DIV_BY_ZERO	2560	Division durch Null
IEEE_OVERFLOW	2816	Überlauf
IEEE_UNDERFLOW	2816	Unterlauf
IEEE_INEXACT	3328	Nicht exakte Operation
IEEE_CONTINUE	64	Zur Programmfortsetzung
IEEE_ALL	0	Zum Rücksetzen aller IEEE–flags
MAXREAL	9.99 ... 99E+255	Größte positive real–Zahl
MINREAL	1.00 ... 00E–267	Kleinste positive real–Zahl

Die ersten sieben integer–Konstanten aus obiger Tabelle werden zur Fehler- und Ausnahmebehandlung benötigt, die in einem der nächsten Abschnitte beschrieben wird.

8.1.2.3.2 Speicherformat für real/Longreal-Konstanten

Wir betrachten zunächst eine real–Konstante , für deren Speicherung 8 Byte mit zusammen 64 bits benötigt werden:

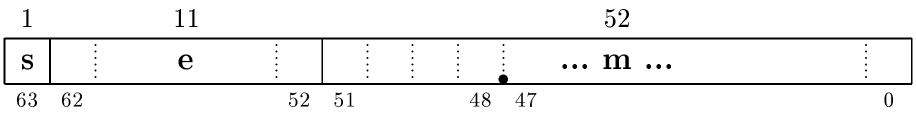


Abbildung 8.1: *Speicherformat einer real–Konstanten*

Die real–Zahl $r = 0$ wird gespeichert, indem alle 64 bits auf Null gesetzt werden. Für $r \neq 0$ bestimmt das 63.-te bit das Vorzeichen von r :

$$s = \begin{cases} 0 & \text{falls } r > 0 \\ 1 & \text{falls } r < 0 \end{cases}$$

Die vorzeichenlose integer–Zahl e ist definiert durch das 52. bis 62. bit, und die Mantisse m wird in den 52 restlichen bits gespeichert, wobei jede BCD–Ziffer der Mantisse 4 bits belegt, so daß insgesamt 13 BCD-Ziffern zur Verfügung stehen. Die Ziffer 9 wird beispielsweise durch die bit–Folge $1\,0\,0\,1$ repräsentiert. Der Dezimalpunkt ist (gedanklich) stets zwischen dem 47. und 48. bit zu setzen. Nach diesen Vereinbarungen gilt für $r \neq 0$:

$$r = (-1)^s \cdot m \cdot 10^{e-255}; \qquad 0 \leq e \leq 510$$

Für normalisierte real–Zahlen gilt: $1 \le m < 10$;
Für denormalisierte real–Zahlen gilt: $10^{-12} \le m \le 0.999999999999$;

Eine **Longreal**–Zahl benötigt 12 Byte und wird ganz analog gespeichert:

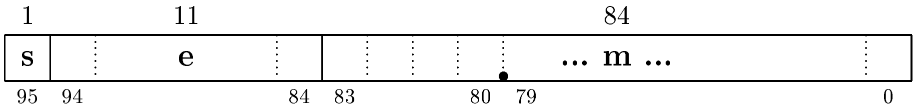


Abbildung 8.2: *Speicherformat einer Longreal–Konstanten*

Die Longreal–Zahl $Lr = 0$ wird gespeichert, indem alle 96 bits auf Null gesetzt werden.
Für $Lr \neq 0$ bestimmt das 95.-te bit das Vorzeichen von Lr :

$$s = \begin{cases} 0 & \text{falls } Lr > 0 \\ 1 & \text{falls } Lr < 0 \end{cases}$$

Die vorzeichenlose integer–Zahl e ist definiert durch das 84. bis 94. bit, und die Mantisse m wird in den 84 restlichen bits gespeichert, wobei jede BCD–Ziffer der Mantisse 4 bits belegt, so daß insgesamt 21 BCD-Ziffern zur Verfügung stehen. Der Dezimalpunkt ist (gedanklich) stets zwischen dem 79. und 80. bit zu setzen. Nach diesen Vereinbarungen gilt für $Lr \neq 0$ wieder:

$$r = (-1)^s \cdot m \cdot 10^{e-511}; \qquad 0 \le e \le 1022$$

Für normal. Longreal–Zahlen gilt: $1 \le m < 10$;
Für denormal. Longreal–Zahlen gilt: $10^{-20} \le m \le 0.99999999999999999999$;

8.1.2.3.3 Hexadezimale Ein-/Ausgabe

Die Darstellung des Speichers für **real**- und **Longreal**-Konstanten in hexadezimaler Form entsprechend der Abbildungen 8.1,8.2 wird ermöglicht durch die Prozeduren

procedure **read** (var f : text; var r : real);
procedure **write** (var f : text; var r : real);

procedure **read** (var f : text; var Lr : Longreal);
procedure **write** (var f : text; var Lr : Longreal);

indem an r bzw. Lr einen Doppelpunkt mit nachfolgendem 'x' oder 'X' angehängt wird. Mit den Wertzuweisungen

```
Lr := -1.23E-33;   { Lr : Longreal }
writeln(Lr:'X');   write(Lr:'x');
```

erhält man die Bildschirmausgabe:

9 D E 1 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9 d e 1 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

'X' bewirkt demnach die Ausgabe der Hexadezimal–Zahlen in Großbuchstaben. Die Hexadezimalziffer 9 beschreibt die bit–Folge 1 0 0 1 mit den Nummern 95 bis 92, und D beschreibt die bit–Folge 1 1 0 1 mit den Nummern 91 bis 88. 9 D E definieren demnach die bit–Folge: 1 0 0 1 1 1 0 1 1 1 1 0 , woraus sich nach Abb. 8.2 für **e** der Wert 478 ergibt; $478 - 511 = -33$ liefert dann den Zehnerexponenten -33 unseres Beispiels.

Ganz entsprechend kann man mit `read(Lr:'X')` und Eingabe der obigen 24-stelligen Hexadezimalfolge der Longreal–Variablen Lr den Wert $-1.23 \cdot 10^{-33}$ zuweisen, wobei zwischen 'X' und 'x' nicht unterschieden wird. Für die Ein-/Ausgabe von **real**–Konstanten in 16-stelliger hexadezimaler Form gelten ganz analoge Aussagen.

8.1.2.3.4 Mantisse/Exponent

Die Standardfunktionen **comp, mant, expo** beziehen sich auf die Form:

$$x = \mathrm{mant}(x) \cdot 10^{\mathrm{expo}(x)}, \quad \text{mit:} \quad 0.1 \leq \mathrm{mant}(x) < 1;$$

Die nachfolgenden Funktionen

function **x_comp** (m : real; ex : integer) : real;
function **x_expo** (r : real) : integer;
function **x_mant** (r : real) : real;

function **x_comp** (m : Longreal; ex : integer) : Longreal;
function **x_expo** (Lr : Longreal) : integer;
function **x_mant** (Lr : Longreal) : Longreal;

des Moduls **x_real** beziehen sich auf eine Zahlendarstellung der Form:

$$x = \mathrm{x\_mant}(x) \cdot 10^{\mathrm{x\_expo}(x)}, \quad \text{mit:} \quad 1 \leq \mathrm{x\_mant}(x) < 10;$$

8.1.2.3.5 Klassifizierung von real/Longreal–Werten

Die Klassifizierung von **real/Longreal**–Konstanten erfolgt mit dem Datentyp **x_ccode** und den Funktionen:

function **x_class** (r : real) : x_ccode;
function **x_class** (Lr : Longreal) : x_ccode;
function **x_value** (c : x_ccode) : real;
function **x_Lvalue** (c : x_ccode) : Longreal;

Der nachfolgend definierte Typ **x_ccode** unterscheidet dabei 10 verschiedene Klassen:

```
 type x_ccode = ( x_sNaN,  { signaling NaN         }
  x_qNaN,  { quiet NaN           }
  x_minf,  { minus infinity      }
  x_mnor,  { negative normalized   }
  x_mden,  { negative denormalized }
  x_mnul,  { minus zero           }
  x_pnul,  { plus zero            }
  x_pden,  { positive denormalized }
  x_pnor,  { positive normalized   }
  x_pinf,  { plus infinity        }
);
```

Mit c: x_ccode liefert die Anweisung c := x_class(r) zum real/Longreal–Wert r den entsprechenden c-Wert vom Typ x_ccode.

Die Funktion x_value liefert zu jedem der 10 Argumente c: x_ccode einen **speziellen** real-Wert; die hexadezimalen Darstellungen dieser real–Konstanten sind zusammen mit ihrem jeweiligen Dezimalwert in folgender Tabelle zusammengestellt:

function x_value (c : x_ccode) : real		
c	**Hexadezimalwert**	**Dezimalwert**
x_sNaN	1 F F 8 0 0 0 0 F F F F F F F F	qNaN
x_qNaN	1 F F 0 0 0 0 0 F F F F F F F F	qNaN
x_minf	9 F F 0 0 0 0 0 0 0 0 0 0 0 0 0	$-$infinity
x_mnor	9 F E 9 9 9 9 9 9 9 9 9 9 9 9 9	$-9.99\ldots99 \cdot 10^{+255}$
x_mden	8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	$-1.00\ldots00 \cdot 10^{-267}$
x_mnul	8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$-0.00\ldots00$
x_pnul	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$+0.00\ldots00$
x_pden	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	$+1.00\ldots00 \cdot 10^{-267}$
x_pnor	1 F E 9 9 9 9 9 9 9 9 9 9 9 9 9	$+9.99\ldots99 \cdot 10^{+255}$
x_pinf	1 F F 0 0 0 0 0 0 0 0 0 0 0 0 0	+infinity

Ganz entsprechend liefert die Funktion x_Lvalue zu jedem der 10 Argumente c: x_ccode einen **speziellen** Longreal-Wert; die hexadezimalen Darstellungen dieser Longreal–Konstanten sind zusammen mit ihrem jeweiligen Dezimalwert in folgender Tabelle zusammengestellt:

function x_Lvalue (c : x_ccode) : Longreal		
c	**Hexadezimalwert**	**Dezimalwert**
x_sNaN	3 F F 8 0 0 0 0 F F F F F F F F 0 0 0 0 0 0 0 0	qNaN
x_qNaN	3 F F 0 0 0 0 0 F F F F F F F F 0 0 0 0 0 0 0 0	qNaN
x_minf	B F F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	−infinity
x_mnor	B F E 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9	$-9.99\ldots99 \cdot 10^{+511}$
x_mden	8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	$-1.00\ldots00 \cdot 10^{-531}$
x_mnul	8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$-0.00\ldots00$
x_pnul	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$+0.00\ldots00$
x_pden	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	$+1.00\ldots00 \cdot 10^{-531}$
x_pnor	3 F E 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9	$+9.99\ldots99 \cdot 10^{+511}$
x_pinf	3 F F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	+infinity

8.1.2.3.6 IEEE Ausnahmebehandlungs–Routinen

Zusammen mit den im ersten Abschnitt definierten Konstanten stehen die nachfolgenden Funktionen und Prozeduren zur Ausnahmebehandlung zur Verfügung:

```
procedure IEEE_environment( action  : integer;
    handler : integer;
    mode    : boolean);
procedure IEEE_trap_enable( handler : integer;
    mode    : boolean);
function  IEEE_test    ( handler : integer) : boolean;
procedure IEEE_set     ( handler : integer);
procedure IEEE_reset   ( handler : integer);
procedure IEEE_save    ( var x   : integer);
procedure IEEE_restore ( x       : integer);
```

Eine genaue Beschreibung findet man in [7]; beachten Sie dort auch das Programmbeispiel im Anhang D.

8.1.2.4 IOSTD, X_INTG, X_STRG

Die Beschreibung der Module **iostd, x_intg, x_strg** findet man in [7].

8.1.2.5 LI_ARI

Das Modul **Li_ari** stellt für den Datentyp **Longreal** alle für die reelle Intervallrechnung notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existieren Operatoren und Prozeduren zur exakten Akkumulator–Auswertung von Longreal–Intervallausdrücken. Eine genaue Beschreibung findet man ab Seite 81.

8.1.2.6 I_ARI

Das Modul **i_ari** stellt für den Typ **real** alle die für die reelle Intervallrechnung notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von Intervallausdrücken. Eine genaue Beschreibung findet man ab Seite 75.

8.1.2.7 RRGI_ARI

Das Modul **rrGi_ari** stellt für den 26-stelligen Datentyp **rrG** alle die für die Implementierung reeller Intervall–Standardfunktionen notwendigen Operatoren und Prozeduren zur Verfügung. Darüber hinaus existieren Hilfsfunktionen zur Wertebereichseinschließung von monotonen Funktionen oder Funktionen mit einem oder mehreren lokalen Extrema. Eine genaue Beschreibung findet man ab Seite 93. Die Intervall–Standardfunktionen sind in Tabelle 4.8 auf Seite 94 zusammengestellt.

8.1.2.8 C_ARIAUX

c_ariaux wird nur vom System–Modul **c_ari** importiert und exportiert zwei Routinen zur Realisierung der komplexen Division für Punktargumente:

1. procedure **produkt** (a,b,c,d : real; var overfl : boolean;
   var p1 : real; var p2 : interval);

   Mit der Definition:

   $$ex := \begin{cases} -267, & \text{falls overfl } = +1 \\ +0, & \text{falls overfl } = +0 \\ +267, & \text{falls overfl } = -1 \end{cases} \qquad \text{gilt für alle a,b,c,d : real}$$

   $$(a \cdot b + c \cdot d) \cdot 10^{ex} \in (p1 + p2)$$

   $p1$: maximalgenaue Näherung für: $(a \cdot b + c \cdot d) \cdot 10^{ex}$

   $p2$: maximalgenaue Einschließung für: $(a \cdot b + c \cdot d - p1) \cdot 10^{ex}$

   Die Berechnung von $p1, p2$ erfolgt wegen des Skalierungsfaktors 10^{ex} ohne Programmabbruch durch internen Über- oder Unterlauf!

2. function **quotient** (z1 : real; z2 : interval; n1 : real; n2 : interval;
   round, zoverfl, noverfl : integer) : real;

   Mit der Anweisung

   ```
   y := quotient (z1,z2,n1,n2,rd,zo,no)
   ```

   und zo,no $\in \{-1, 0, +1\}$ wird intern zunächst durch staggered correction Technik unter der Voraussetzung: $(n1 + n2) > 0$ eine Einschließung Q des exakten Intervalls

   $$q := \frac{(z1 + z2)}{(n1 + n2)} \cdot 10^{(zo-no)\cdot 267} \subseteq \mathrm{Q}$$

berechnet, wobei nur Überlauf entstehen kann. Der Funktionswert y: real wird dann durch den Rundungsparameter rd wie folgt bestimmt:

$$\text{y} := \begin{cases} \text{Nächstkleinere real-Zahl bzgl. Q.inf} \le \text{q.inf}, & \text{falls rd} = -1 \\ \text{Nächste real-Zahl bzgl. 'Mittelpunkt' von Q}, & \text{falls rd} = +0 \\ \text{Nächstgrößere real-Zahl bzgl. Q.sup} \ge \text{q.sup}, & \text{falls rd} = +1 \end{cases}$$

Anmerkungen:

- Da die Einschließung Q mittels staggered correktion berechnet wird, ist Q eine sehr enge Einschließung des exakten Quotienten q . Deshalb liefert z.B. rd $= -1$ mit y fast immer die größte Unterschranke von q, und nur in extremen Ausnahmesituationen liegt zwischen y und q.inf eine weitere real–Zahl. Für rd $= +1$ gelten ganz entsprechende Aussagen. Da bei der komplexen Division die gerichteten Rundungen mittels rd $= \pm 1$ berechnet werden, muß deshalb die Fehlerschranke größer sein als $1 \cdot 10^{-12}$ und wird festgelegt durch $1.5 \cdot 10^{-12}$; vergl. die Tabelle auf Seite 108.
- Wir setzen voraus, daß die Intervalle $(z1 + z2), (n1 + n2)$ selbst, z.B. mit der Prozedur produkt(...), durch staggered correction Technik berechnet werden, so daß ihre Durchmesser extrem klein gegenüber dem Abstand benachbarter real–Zahlen angenommen werden, und dies gilt dann auch für die Einschließung Q des exakten Quotienten q . Enthält nun Q in einer seltenen Ausnahmesituation den Mittelpunkt seiner umgebenden real–Zahlen, so können unterschiedliche Zahlen aus Q verschiedene (benachbarte) 'nächste real–Zahlen' besitzen, so daß dann der oben genannte 'Mittelpunkt' von Q nur eine der beiden möglichen 'nächsten real–Zahlen' festlegt. Da bei der komplexen Division der Operator / durch die Funktion quotient(..) mit rd $= 0$ definiert wird, ist seine Fehlerschranke nicht mehr $5 \cdot 10^{-13}$ sondern $1 \cdot 10^{-12}$, d.h. die komplexe Division ist nur hochgenau; vergl. die Tabelle auf Seite 108.

8.1.2.9 LC_HELP

Lc_help wird nur vom System–Modul **Lc_ari** importiert und exportiert zwei Routinen zur Realisierung der komplexen Division für Punktargumente:

1. procedure **produkt** (a,b,c,d : Longreal; var overfl : boolean;
   var p1 : Longreal; var p2 : Linterval);

   Mit der Definition:

$$ex := \begin{cases} -531, & \text{falls overfl} = +1 \\ +0, & \text{falls overfl} = +0 \\ +531, & \text{falls overfl} = -1 \end{cases} \qquad \text{gilt für alle a,b,c,d : Longreal}$$

$$(a \cdot b + c \cdot d) \cdot 10^{ex} \in (p1 + p2)$$

$p1$: maximalgenaue Näherung für: $(a \cdot b + c \cdot d) \cdot 10^{ex}$

$p2$: maximalgenaue Einschließung für: $(a \cdot b + c \cdot d - p1) \cdot 10^{ex}$

Die Berechnung von $p1, p2$ erfolgt wegen des Skalierungsfaktors 10^{ex} ohne Programmabbruch durch internen Über- oder Unterlauf!

2. function **quotient** (z1 : Longreal; z2 : Linterval; n1 : Longreal; n2 : Linterval; round, zoverfl, noverfl : integer) : Longreal;

   Mit der Anweisung

```
y := quotient (z1,z2,n1,n2,rd,zo,no)
```

   und zo,no $\in \{-1, 0, +1\}$ wird intern zunächst durch staggered correction Technik unter der Voraussetzung: $(n1 + n2) > 0$ eine Einschließung Q des exakten Intervalls

$$q := \frac{(z1 + z2)}{(n1 + n2)} \cdot 10^{(zo-no)\cdot 531} \subseteq \mathrm{Q}$$

   berechnet, wobei nur Überlauf entstehen kann. Der Funktionswert y: Longreal wird dann durch den Rundungsparameter rd wie folgt bestimmt:

$$\mathrm{y} := \begin{cases} \text{Nächstkleinere Longreal-Zahl bzgl. Q.inf} \le \text{q.inf}, & \text{falls rd} = -1 \\ \text{Nächste Longreal-Zahl bzgl. 'Mittelpunkt' von Q ,} & \text{falls rd} = +0 \\ \text{Nächstgrößere Longreal-Zahl bzgl. Q.sup} \ge \text{q.sup}, & \text{falls rd} = +1 \end{cases}$$

   Anmerkungen:

   - Da die Einschließung Q mittels staggered correktion berechnet wird, ist Q eine sehr enge Einschließung des exakten Quotienten q . Deshalb liefert z.B. rd $= -1$ mit y fast immer die größte Unterschranke von q, und nur in extremen Ausnahmesituationen liegt zwischen y und q.inf eine weitere Longreal–Zahl. Für rd $= +1$ gelten ganz entsprechende Aussagen. Da bei der komplexen Division die gerichteten Rundungen mittels rd $= \pm 1$ berechnet werden, muß deshalb die Fehlerschranke größer sein als $1 \cdot 10^{-20}$ und wird festgelegt durch $1.5 \cdot 10^{-20}$; vergl. die Tabelle auf Seite 113.
   - Wir setzen voraus, daß die Intervalle $(z1 + z2), (n1 + n2)$ selbst, z.B. mit der Prozedur produkt(...), durch staggered correction Technik berechnet werden, so daß ihre Durchmesser extrem klein gegenüber dem Abstand benachbarter Longreal–Zahlen angenommen werden, und dies gilt dann auch für die Einschließung Q des exakten Quotienten q . Enthält nun Q in einer seltenen Ausnahmesituation den Mittelpunkt seiner umgebenden Longreal–Zahlen, so können unterschiedliche Zahlen aus Q verschiedene (benachbarte) 'nächste Longreal–Zahlen' besitzen, so daß dann der oben genannte 'Mittelpunkt' von Q nur eine der beiden möglichen 'nächsten Longreal–Zahlen' festlegt. Da bei der komplexen Division der Operator / durch die Funktion quotient(..) mit rd $= 0$ definiert wird, ist seine Fehlerschranke nicht mehr $5 \cdot 10^{-21}$ sondern $1 \cdot 10^{-20}$, d.h. die komplexe Division ist nur hochgenau; vergl. die Tabelle auf Seite 113.

8.1.2.10 LC_ARI

Das Modul **Lc_ari** stellt für den Datentyp **Longreal** alle für komplexe Rechnungen notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existieren Operatoren und Prozeduren zur exakten Akkumulator–Auswertung von Lcomplex–Ausdrücken.

Für den 26-stelligen Datentyp **rrGcomplex** stehen zusätzlich die wichtigsten Operatoren und Transferfunktionen zur Realisierung 26-stelliger Standardfunktionen zur Verfügung. Eine genaue Beschreibung findet man ab Seite 112.

8.1.2.11 LCI_ARI

Das Modul **Lci_ari** stellt für den Datentyp **Longreal** alle für komplexe Intervallrechnungen notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existieren Operatoren und Prozeduren zur exakten Akkumulator–Auswertung von Lcinterval–Ausdrücken.

Für den 26-stelligen Datentyp **rrGcinterval** stehen zusätzlich die wichtigsten Operatoren und Transferfunktionen zur Realisierung 26-stelliger Intervall–Standardfunktionen zur Verfügung. Eine genaue Beschreibung findet man dazu ab Seite 136.

8.1.2.12 C_ARI

Das Modul **c_ari** stellt für den Typ **real** alle die für komplexe Rechnungen notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von complex–Ausdrücken. Eine genaue Beschreibung findet man ab Seite 107.

8.1.2.13 CI_ARI

Das Modul **ci_ari** stellt für den Typ **real** alle die für komplexe Intervallrechnungen notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von cinterval–Ausdrücken. Genaue Beschreibungen findet man ab Seite 127.

8.1.2.14 MV_ARI

In diesem Modul werden zum Basis–Typ **real** die für das Rechnen mit reellen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von rmatrix- oder rvector–Ausdrücken. Genaue Beschreibungen findet man ab Seite 161.

8.1.2.15 MVI_ARI

In diesem Modul werden zum Basis–Typ **real** die für das Rechnen mit reellen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von imatrix- oder ivector–Ausdrücken. Eine genaue Beschreibung findet man ab Seite 172.

8.1.2.16 MVC_ARI

Das Modul mvc_ari stellt zum Basis–Typ **real** die für das Rechnen mit komplexen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren zur Verfügung. Darüber hinaus existiert auch das komplette Lattenkreuzkonzept zur exakten Auswertung von cmatrix- oder cvector–Ausdrücken. Eine genaue Beschreibung findet man ab Seite 186.

8.1.2.17 MVCI_ARI

In diesem Modul werden zum Basis–Typ **real** die für das Rechnen mit komplexen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Darüber hinaus existiert das komplette Lattenkreuzkonzept zur exakten Auswertung von cimatrix- oder civector–Ausdrücken. Eine genaue Beschreibung findet man ab Seite 198.

8.1.2.18 LMV_ARI

In diesem Modul werden zum Basis–Typ **Longreal** die für das Rechnen mit reellen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Operanden vom Basis–Typ **real** sind nicht erlaubt. Darüber hinaus existieren alle notwendigen Operatoren zur exakten Auswertung von Lrmatrix- oder Lrvector–Ausdrücken, wobei auch rmatrix- bzw. rvector–Operanden möglich sind. Eine genaue Beschreibung findet man ab Seite 165.

8.1.2.19 LMVI_ARI

In diesem Modul werden zum Basis–Typ **Longreal** die für das Rechnen mit reellen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Operanden vom Basis–Typ **real** sind nicht erlaubt. Darüber hinaus existieren alle notwendigen Operatoren zur exakten Auswertung von Limatrix- oder Livector–Ausdrücken, wobei in diesen Ausdrücken auch rmatrix- und imatrix- bzw. rvector- und ivector–Operanden möglich sind. Eine genaue Beschreibung findet man ab Seite 178.

8.1.2.20 LMVC_ARI

In diesem Modul werden zum Basis–Typ **Longreal** die für das Rechnen mit komplexen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Operanden vom Basis–Typ **real** sind nicht erlaubt. Zur exakten Auswertung von Lcmatrix- oder Lcvector–Ausdrücken stehen die entsprechenden Prozeduren aus dem Modul **Lc_ari** zur Verfügung. Eine genaue Beschreibung mit Beispielen findet man ab Seite 191.

8.1.2.21 LMVCI_ARI

In diesem Modul werden zum Basis–Typ **Longreal** die für das Rechnen mit komplexen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt. Operanden vom Basis–Typ **real** sind nicht erlaubt. Zur exakten Auswertung von Lcimatrix- oder Lcivector–Ausdrücken stehen die entsprechenden Prozeduren aus dem Modul **Lci_ari** zur Verfügung. Eine genaue Beschreibung mit Beispielen findet man ab Seite 207.

8.2 Hilfsmodule

In Form einer zusätzlichen Modulbibliothek stehen **PASCAL-XSC** -Routinen der BCD–Version für die Lösung häufig auftretender numerischer Probleme zur Verfügung. Die darin verwendeten Lösungsmethoden berechnen eine scharfe Einschließung der wahren Lösung des gestellten Problems und weisen gleichzeitig die Existenz und Eindeutigkeit im angegebenen Intervall nach. Diese Routinen haben daher folgende Vorzüge:

- Die Berechnung einer Lösung erfolgt mit maximaler oder hoher, immer aber mit kontrollierter Genauigkeit, auch in vielen schlecht konditionierten Fällen.
- Die Richtigkeit des Ergebnisses wird verifiziert, d.h. es wird eine Einschließungsmenge berechnet, in der die Existenz und Eindeutigkeit der exakten Lösung gesichert ist.
- Falls keine Lösung existiert oder das Problem zu schlecht konditioniert ist, so wird eine Fehlermeldung ausgegeben.
- Der Benutzer kann kaum etwas falsch machen, und somit sind diese Methoden auch von Nichtspezialisten anwendbar.
- Bei der BCD–Version entfallen die bei Binärversionen oft so lästigen Konversionsprobleme, da die über die Tastatur eingegebenen, dezimalen Gleitpunkt–Zahlen **fehlerfrei** gespeichert werden und so den Anwender–Routinen unverfälscht zur Verfügung stehen. Beachten Sie bitte, daß bei einer Binärversion die Dezimalzahl 0.1 schon nicht mehr fehlerfrei gespeichert wird!

Im einzelnen decken die **PASCAL-XSC** Routinen die folgenden Gebiete ab:

- Lineare Gleichungssysteme
  - Vollbesetzte Systeme
  - Matrixinvertierung
  - Ausgleichsprobleme
  - Berechnung der Pseudoinversen
  - Bandmatrizen
  - Spärlich besetzte Matrizen
- Polynomauswertung
  - in einer Variablen
  - in mehreren Variablen
- Polynomnullstellen
- Eigenwerte und Eigenvektoren
  - Symmetrische Matrizen
  - Beliebige Matrizen

- Anfangs- und Randwertprobleme bei gewöhnlichen Differentialgleichungen
  - linear
  - nichtlinear
- Auswertung von arithmetischen Ausdrücken
- Nichtlineare Gleichungssysteme
- Numerische Quadratur
- Integralgleichungen
- Automatische Differentiation
- Spezielle Funktionen der mathematischen Physik

Näheres zu den einzelnen Routinen und Modulen kann der begleitenden Dokumentation der **PASCAL-XSC** Numerikbibliothek entnommen werden.

Über die jeweils behandelte Grundproblematik hinaus sind die Einsatzmöglichkeiten dieser Routinen vielfältig. Mit ihnen wird die Beantwortung so interessanter und wichtiger Fragestellungen möglich, wie z.B.

- Feststellung der Kondition von Problemen durch Intervalleingabe.
- Feststellung von lokalen Ausschlußgebieten von Lösungen, indem man große und kleine Eindeutigkeitsbereiche bestimmt, deren Differenzmenge dann das Ausschlußgebiet darstellt.
- Verifikation von Aussagen wie Bestimmung des Mindestranges einer Matrix oder Bestimmung einer Kugel bzw. einer Halbebene, in der alle Nullstellen eines komplexen Polynoms liegen. Damit ist es z.B. möglich, die Stabilität elektotechnischer Schaltungen zu garantieren, soweit das Modell die Realität erfaßt.
- Parameterkontrolle bei Modellbildungen. Es kann festgestellt werden, welche Modelldaten mit welcher Empfindlichkeit in eine Modellformel einwirken und umgekehrt, wie genau Meßdaten sein müssen, damit eine abhängige Größe überhaupt noch eine vorgeschriebene Genauigkeitsrelevanz besitzt.

Die Modulbibliothek soll den Anwender in die Lage versetzen, eigene Problemstellungen so zu lösen, daß seine numerischen Programmergebnisse wirkliche mathematische Aussagekraft besitzen und nicht nur eine Näherung liefern, über deren Genauigkeit allenfalls spekuliert werden kann nach dem Motto:

'Der Rechner liefert eine 16-stellige Lösung, von denen die ersten vier Ziffern hoffentlich wieder richtig sind'

Diese heute noch weit verbreitete Denkweise auch bei Mathematikern sollte mit Hilfe der jetzt vorhandenen **PASCAL-XSC** Werkzeuge bald überwunden sein!

Literaturverzeichnis

[1] G. Alefeld, J. Herzberger. – *Einführung in die Intervallrechnung.* Reihe Informatik/12; BI, 1974; ISBN 3-411-01466-0;

[2] F. Blomquist. – *Automatische a priori Fehlerabschätzungen.* Inst. f. Angewandte Mathematik, Universität Karlsruhe, 1992.

[3] R. Hammer, M. Hocks, U. Kulisch, D. Ratz. – *Numerical Toolbox for Verified Computing I.* Springer Series in Computational Mathematics 21.

[4] R. Hammer, M. Hocks, U. Kulisch, D. Ratz. – *Numerical Toolbox for Verified Computing II.* Springer Series in Computational Mathematics 21.

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

[6] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich **PASCAL-XSC** – *Language Reference with Examples.* Springer-Verlag, New York, 1992.

[7] Numeric Software GmbH: **PASCAL-XSC** *: A PASCAL Extension for Scientific Computation. User's Guide.* Numeric Software GmbH, P.O.Box 2232, D-76492, Baden-Baden, Germany, 1991.

Stichwortverzeichnis