R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich

PASCAL–XSC

Sprachbeschreibung mit Beispielen

PASCAL–XSC ist eine universelle PASCAL–Erweiterung mit umfangreichen Standardmodulen für wissenschaftliches Rechnen. Sie ist verfügbar für Personal Computer, Workstations, Großrechner und Supercomputer mittels einer Implementierung in C.

PASCAL–XSC vereinfacht das Programmieren im technisch-wissenschaftlichen Bereich durch modularen Programmaufbau, Definition eigener Operatoren, Überladen von Funktionen, Prozeduren und Operatoren, Funktionen und Operatoren mit allgemeinem Ergebnistyp, dynamische Felder, Standardarithmetikmodule für zusätzliche numerische Datentypen mit Operatoren von höchster Genauigkeit, hochgenaue Standardfunktionen und exakte Ausdrucksauswertung.

Zu PASCAL–XSC sind zahlreiche numerische Problemlöseroutinen mit automatischer Verifikation des Ergebnisses verfügbar. Die Sprache unterstützt die Entwicklung solcher Routinen.

R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich

PASCAL–XSC

Sprachbeschreibung mit Beispielen

PASCAL–XSC ist eine universelle PASCAL–Erweiterung mit umfangreichen Standardmodulen für wissenschaftliches Rechnen. Sie ist verfügbar für Personal Computer, Workstations, Großrechner und Supercomputer mittels einer Implementierung in C.

PASCAL–XSC vereinfacht das Programmieren im technisch-wissenschaftlichen Bereich durch modularen Programmaufbau, Definition eigener Operatoren, Überladen von Funktionen, Prozeduren und Operatoren, Funktionen und Operatoren mit allgemeinem Ergebnistyp, dynamische Felder, Standardarithmetikmodule für zusätzliche numerische Datentypen mit Operatoren von höchster Genauigkeit, hochgenaue Standardfunktionen und exakte Ausdrucksauswertung.

Zu PASCAL–XSC sind zahlreiche numerische Problemlöseroutinen mit automatischer Verifikation des Ergebnisses verfügbar. Die Sprache unterstützt die Entwicklung solcher Routinen.

Vorwort

Das vorliegende Buch beschreibt eine PASCAL-Erweiterung für wissenschaftliches Rechnen mit dem Kurztitel PASCAL–XSC (PASCAL eXtension for Scientific Computation). Die Sprache ist das Ergebnis langjähriger Bemühungen von Mitarbeitern des Instituts für Angewandte Mathematik der Universität Karlsruhe und einiger ihm verbundener Wissenschaftler, den Rechner arithmetisch wesentlich mächtiger zu machen als dies bislang der Fall war und das Programmieren von Algorithmen ein erhebliches Stück näher an die in der Mathematik übliche Notation heranzuführen. Zudem unterstützt die Sprache PASCAL–XSC die Entwicklung von selbstverifizierenden Algorithmen, die durch Anwendung mathematischer Fixpunktsätze eine automatische Ergebnisverifikation durch den Rechner selbst ermöglichen.

Die Arbeiten begannen bereits Ende der sechziger Jahre mit der Entwicklung einer allgemeinen Theorie der Rechnerarithmetik. Als erste Folge ergab sich die Notwendigkeit der Entwicklung und Implementierung neuartiger Algorithmen zur Realisierung der arithmetischen Verknüpfungen. Etwa im Jahre 1975 begannen dann die Arbeiten mit dem Entwurf geeigneter Programmiersprachen mit Vorstudien zur Implementierung zunächst auf der Basis von PASCAL, dann aber auch als Erweiterung von FORTRAN. Der Bau von Übersetzern schloß sich unmittelbar an. Etwa seit dem Jahre 1980 wurden dann auch systematisch Algorithmen für Grundaufgaben der Numerik mit automatischer Ergebnisverifikation entwickelt.

Eine große Zahl von Mitarbeitern sowie dem Institut verbundener Kollegen hat durch wertvolle Diskussionsbeiträge, aber auch durch langjährige Mitarbeit oder sonstige Förderung zu dem jetzt vorliegenden Stande beigetragen. Die wesentlichen Stützen dieser Entwicklung seien deshalb hier aufgelistet: U. Allendörfer, H. Bleher, H. Böhm, G. Bohlender, K. Braune, D. Claudio, D. Cordes, A. Davidenkoff, H. C. Fischer, S. Geörg, K. Grüner, R. Hammer, E. Kaucher, R. Kelch, R. Kirchner, R. Klatte, W. Klein, W. Krämer, U. Kulisch, R. Lohner, M. Metzger, W. L. Miranker, M. Neaga, L. B. Rall, D. Ratz, S. M. Rump, R. Saier, D. Shiriaev, L. Schmidt, G. Schumacher, Ch. Ullrich, W. Walter, M. Weichelt, H. W. Wippermann und J. Wolff von Gudenberg. Ihnen allen sei für ihre Mitwirkung auch an dieser Stelle noch einmal aufrichtig und herzlich gedankt. Besonderer Dank gebührt darüberhinaus auch den vielen Studenten, die durch frühzeitige Benutzung zur Stabilität von Sprache und Compiler beigetragen haben.

Im vorliegenden Band wird die Sprache PASCAL–XSC vollständig beschrieben. Dabei ist der zu Standard PASCAL gehörende Teil nur knapp gefaßt. Die durch PASCAL–XSC gekennzeichneten Erweiterungen sind hingegen ausführlich dargestellt. Zur praktischen Verwendung und zum leichteren Kennenlernen und Vertrautwerden mit den neuen Sprachelementen wurde ein ausführlicher Abschnitt mit Übungsaufgaben und Lösungen aufgenommen. Ein kompletter Satz von Syntaxdiagrammen, sowie ausführliche Register und Verzeichnisse schließen das Handbuch ab.

Zum Schluß sei noch erwähnt, daß auch uns bekannt ist, daß eine Programmiersprache niemals abgeschlossen und vollkommen und stets noch verbesserungsfähig ist. Das Hauptanliegen bei der Entwicklung der vorliegenden Sprache bestand darin, ein für numerische Anwendungen im technisch-wissenschaftlichen Bereich geeignetes Werkzeug zur Verfügung zu stellen. Wohlwollende kritische Bemerkungen und Verbesserungsvorschläge nehmen wir gerne entgegen.

Karlsruhe, im November 1990 — Autoren und Herausgeber

Zur Entstehung dieses Buches

Das vorliegende Handbuch wurde vollständig mit dem Textsystem LaTeX bzw. TeX auf *IBM PS/2 Model 70* unter *DOS 4.0* erstellt. Die wesentliche Phase der Zusammenstellung von Texten, das Entwerfen der notwendigen *Makros* und *Environments*, die Herstellung der Syntaxdiagramme, die Einarbeitung der Korrekturen, die abschließende Endherstellung mit dem Erstellen von Anhängen und Verzeichnissen sowie auch die Gestaltung des endgültigen Layouts wurde von Co-Autor Dietmar Ratz durchgeführt.

Die Autoren

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Beschreibung der Sprache PASCAL–XSC besteht neben Einleitung und Anhang im wesentlichen aus den drei großen Kapiteln *Sprachbeschreibung, Arithmetikmodule* und *Übungsaufgaben* und einem nur knapp gehaltenen Kapitel über *Problemlöseroutinen*.

In Kapitel 1 (Einleitung) wird, nach Hinweisen zu den verwendeten Schreibweisen, in kurzen Abschnitten ein Einblick in die historische Entwicklung von PASCAL–XSC, die axiomatische Definition der Rechnerarithmetik und deren Einbettung in Programmiersprachen gegeben. Der letzte Abschnitt bringt schließlich eine kurze Übersicht über die Sprache PASCAL–XSC.

Kapitel 2 (Sprachbeschreibung) enthält die formale Sprachdefinition, wobei der PASCAL-Standard nur knapp, die Erweiterungen von PASCAL–XSC ausführlich wiedergegeben werden.

Kapitel 3 (Arithmetikmodule) beschreibt die für die zusätzlichen arithmetischen Datentypen von PASCAL–XSC zur Verfügung gestellten Module mit ihren Operatoren, Funktionen und Prozeduren. Das anschließende Kapitel 4 (Problemlöseroutinen) enthält dann eine straffe Zusammenfassung der zur Zeit vorhandenen PASCAL–XSC Routinen zur Lösung häufig auftretender numerischer Probleme, die in Form einer zusätzlichen Modulbibliothek zur Verfügung stehen.

Mit dem abschließenden Kapitel 5 (Übungsaufgaben) erhält der Leser die Möglichkeit, anhand von einfachen Aufgaben mit Lösungen die neuen Sprachelemente praktisch kennenzulernen und zu vertiefen.

Im Anhang finden sich schließlich die Syntaxdiagramme von PASCAL–XSC sowie ausführliche Verzeichnisse der Wortsymbole, Standardnamen, Operatoren, Funktionen und Prozeduren des Sprachkerns und der Arithmetikmodule.

1.1 Typographie

Zur Kennzeichnung bzw. Hervorhebung bestimmter Wörter, Namen oder auch Absätze werden die folgenden Schriftarten verwendet:

Kursivschrift	dient zur Hervorhebung bestimmter Wörter im laufenden Text.

Fettschrift	wird benutzt zur Kennzeichnung der Wortsymbole (wie z. B. **begin**, **module**) im laufenden Text oder in Programmstücken.
Schrägschrift	kennzeichnet Standardnamen (wie z. B. *integer*, *real*) und Bezeichner aus Programmbeispielen im laufenden Text.
`Schreibmaschinenschrift`	wird verwendet für Listings und Ausgabeprotokolle von Programmen, die direkt von der Datei oder der Druckerausgabe übernommen wurden.

Literaturverweise werden stets in der Form [*nr*] mit der Nummer *nr* des entsprechenden Eintrags im Literaturverzeichnis angegeben.

1.2 Historische Entwicklung

Elektronische Rechenanlagen sind für Anwendungen im naturwissenschaftlich-technischen Bereich zur Approximation des Rechnens mit reellen Zahlen üblicherweise mit einer Gleitkommaarithmetik ausgestattet. Alle höheren Programmiersprachen erlauben es, diese Operationen mit den üblichen Operationssymbolen anzusprechen und auch einfache Ausdrücke, Formeln oder Funktionen damit in gewohnter Weise niederzuschreiben. In der Mathematik und den Naturwissenschaften ist jedoch der Begriff der arithmetischen Verknüpfung oder der Funktion keineswegs nur auf die reellen Zahlen beschränkt. Hier gibt es unter anderem Verknüpfungen in den üblichen Vektorräumen und auch vektorwertige Funktionen und es ist ineffektiv, auf dem Rechner diese Begriffe immer auf die elementaren Gleitkommaverknüpfungen zurückzuführen und dann durch schwerfällige Prozeduraufrufe zu realisieren. Zudem werden durch diese Vorgehensweise viele unnötige Rechenungenauigkeiten eingeschleppt.

Am Institut für Angewandte Mathematik der Universität Karlsruhe (Prof. Kulisch) wurde daher bereits in den sechziger Jahren intensiv auf dem Gebiet der Rechnerarithmetik, insbesondere auch mit Intervallarithmetik experimentiert. Dabei stellte sich heraus, daß es für viele Anwendungen äußerst nützlich ist, den Rechner bzgl. der Arithmetik wesentlich mächtiger auszustatten als dies mit der üblichen Gleitkommaarithmetik bislang der Fall war. So entstand die Forderung, jeden technisch-wissenschaftlichen Rechner, ob klein oder groß, zu einem Vektorrechner im mathematischen Sinne[1] zu machen. Das heißt, der Rechner soll alle Operationen in den üblichen Vektorräumen, wie reelle und komplexe Zahlen, reelle und komplexe Vektoren und Matrizen, sowie in den zugehörigen Intervallräumen als Grundverknüpfungen für entsprechend vordefinierte Datentypen mit höchster Genauigkeit direkt bereitstellen, ohne diese über die gegebene Gleitkommaarithmetik zu approximieren. Eine vollständige mathematische Analyse dieser Forderungen war Anfang der siebziger Jahre fertig und führte zu zwei Buchveröffentlichungen ([20], [24]).

[1] Der Begriff Vektorrechner wird häufig synonym für einen mit Pipelineoperationen ausgestatteten Rechner gebraucht. Dies ist hier nicht gemeint.

Algorithmen und schnelle Hardwareschaltungen für die Realisierung dieser Forderungen wurden nach und nach entwickelt und implementiert. Für alle Arten von Rechnern, wie Personal Computer, Workstation, Universalrechner als auch Großrechner und Supercomputer steht heute eine große Auswahl von Realisierungsmöglichkeiten zur Verfügung. Die neuen Verknüpfungen, wie z. B. das Produkt zweier Matrizen, liefern immer ein Ergebnis, welches aus dem exakten Resultat durch höchstens eine einzige Rundung entsteht. Bei Ausführung in gleicher Technologie (Software, Mikrocode, Hardware, Pipelinetechnik) sind die neuen Verknüpfungen aber nicht nur genauer, sondern im allgemeinen sogar schneller als wenn man sie auf herkömmliche Weise über die gegebene Gleitkommaarithmetik simuliert. Von Seiten der Hersteller wurde die Richtigkeit und Nützlichkeit dieser Vorgehensweise nach und nach erkannt, so daß im Laufe der Jahre immer mehr Produkte auf den Markt kamen, welche bereits in der Hardware die neuen Anforderungen an die Arithmetik unterstützen.

Schwierigkeiten ergaben sich allerdings sofort auf dem Gebiete der Programmiersprachen. Herkömmliche Programmiersprachen wie ALGOL, FORTRAN, PL/1, PASCAL oder MODULA erlauben es nicht, ein durch die Hardware unterstütztes maximal genaues Matrixprodukt, eine maximal genaue Multiplikation komplexer Zahlen oder eine Intervallverknüpfung sprachlich mit den üblichen Operationssymbolen anzusprechen. Dies machte eine Weiterentwicklung von Programmiersprachen erforderlich, wobei sich alle Einzelheiten und Details ziemlich zwangsläufig aus den mathematisch-arithmetischen Anforderungen ergaben. So wurde in Zusammenarbeit zweier Institute der Universitäten Karlsruhe und Kaiserslautern (Prof. Kulisch und Prof. Wippermann) in den Jahren 1976 bis 1979 zunächst eine PASCAL-Erweiterung entwickelt und implementiert, welche den Namen PASCAL–SC (PASCAL for Scientific Computation) erhalten hat. In der Folgezeit wurde dann in Zusammenarbeit zwischen der Firma IBM und dem Institut für Angewandte Mathematik der Universität Karlsruhe eine entsprechende FORTRAN 77 Erweiterung entwickelt und für IBM/370 Rechenanlagen am Institut für Angewandte Mathematik der Universität Karlsruhe implementiert. Das Ergebnis wird heute unter dem Namen ACRITH–XSC von IBM als Programmprodukt vertrieben.

ACRITH–XSC enthält einige Konstrukte wie dynamische Felder, Überladen von Funktionsnamen usw., welche in PASCAL–SC nicht enthalten sind. Parallel zur Entwicklung von ACRITH–XSC ist so auch die Sprache PASCAL–XSC entstanden. Sie kann unter anderem mittels eines nach C übersetzenden Compilers und eines in C geschriebenen Laufzeitsystems in nahezu identischer Weise auf allen UNIX-Systemen betrieben werden. Der Benutzer hat dadurch die Möglichkeit, seine Programme etwa auf seinem Personal Computer zu entwickeln und sie dann anschließend mit demselben Compiler auf einer Großrechenanlage laufen zu lassen.

Die Verfügbarkeit von PASCAL–SC, ACRITH–XSC und PASCAL–XSC hat zu einer großen Anzahl neuartiger Problemlösungen geführt. Dabei spielte insbesondere auch die einfache Verfügbarkeit von Intervallverknüpfungen über allen Grundräumen eine wesentliche Rolle. Neben der Möglichkeit, Schranken für die Lösung eines Problems zu berechnen, bringen die Intervalle das Kontinuum auf den Rechner. Eine einzige Auswertung einer Funktion über einem Intervall kann ausreichen, um im mathematisch strengen Sinne festzustellen, daß die Funktion keine Nullstelle in diesem Intervall besitzt. In Fortführung dieser Ideen lassen sich mathematische Fixpunktsätze vom Brouwerschen oder Schauderschen Typ einsetzen, um Existenz- und Eindeutigkeitsaussagen numerischer Probleme mit dem Rechner selbst zu gewinnen oder die Korrektheit eines berechneten Ergebnisses vom Rechner selbst verifizieren zu lassen. So sind beispielsweise für Rand- und Eigenwertprobleme gewöhnlicher Differentialgleichungen und für Systeme von linearen und nichtlinearen Integralgleichungen Programmpakete entwickelt worden, welche sowohl die Existenz und Eindeutigkeit der Lösung nachweisen als auch kontinuierlich enge Schranken für die Lösung selbst berechnen (vgl. [23]). Auf vielen Anwendungsgebieten wie in der Mechanik, der Chemie, bei Problemen des Chaos oder bei der Suche nach periodischen Lösungen von Differentialgleichungen konnten durch Einsatz der neuen Werkzeuge bei numerisch bisher nicht zugänglichen Problemen überraschende Lösungen gefunden werden. Es sei diesbezüglich auf das Literaturverzeichnis verwiesen.

1.3 Definition der Rechnerarithmetik

Bei der Schaffung von Programmiersprachen (ALGOL und FORTRAN) in den fünfziger Jahren galt es als allgemeiner Konsens, daß man die Arithmetik nicht verbindlich festlegt, sondern ihre Realisierung dem Hersteller überläßt. Dies hatte zur Folge, daß der Benutzer im allgemeinen nicht genau weiß, was geschieht, wenn er in einem Programm die Symbole `+`, `-`, `*` oder `/` verwendet. Zwei Rechenmaschinen verschiedener Hersteller unterscheiden sich zudem in der Regel bezüglich der Arithmetik. Die Numerik ist folglich nicht in der Lage, auf allgemein verbindlichen Grundannahmen über die Rechnerarithmetik aufzubauen. Alles was man tun kann, besteht darin, die Grundannahmen, welche bei der Fehleranalyse numerischer Algorithmen benutzt werden, gewissermaßen als Ersatzdefinition für die Rechnerarithmetik heranzuziehen.

Die zunehmende Leistungsfähigkeit und Geschwindigkeit von Rechenanlagen verlangt jedoch, auch die Arithmetik präziser zu definieren. Da auf einer Rechenanlage nur endlich viele Zahlen darstellbar sind, muß die Menge $\mathbb{R}$ der reellen Zahlen auf eine Teilmenge R, die sogenannten Gleitkommazahlen, abgebildet werden. Diese Abbildung $\square : \mathbb{R} \to R$ bezeichnen wir als Rundung wenn sie die beiden folgenden Eigenschaften besitzt:

$$\text{(R1)} \quad \square a = a \quad \text{für alle } a \in R \qquad \text{(Projektion)}$$

$$\text{(R2)} \quad a \le b \Rightarrow \square a \le \square b \quad \text{für alle } a, b \in \mathbb{R}. \qquad \text{(Monotonie)}$$

Eine Rundung mit der Eigenschaft

$$\text{(R3)} \quad \square(-a) = -\square a \quad \text{für alle } a \in \mathbb{R} \qquad \text{(Antisymmetrie)}$$

heißt antisymmetrisch. Es gibt verschiedene, gebräuchliche antisymmetrische Rundungen, wie z. B. die Rundung nach innen (zur Null), die Rundung nach außen oder die Rundung zur nächstgelegenen Gleitkommazahl. Von den approximierenden Verknüpfungen Operationen $\boxplus$, $\boxminus$, $\boxdot$ und $\boxslash$ für Gleitkommazahlen verlangen wir, daß sie die folgende Eigenschaft besitzen:

$$\text{(RG)} \qquad a \boxcirc b = \square(a \circ b) \qquad \text{für alle } a, b \in R \text{ und } \circ \in \{+, -, \cdot, /\}.$$

Dabei bezeichnen $+, -, \cdot, /$ die Verknüpfungen für relle Zahlen.

Eine Abbildung, welche die Eigenschaften (R1), (R2), (R3) und (RG) erfüllt, bezeichnen wir als einen *Semimorphismus.*

Alle durch (RG), (R1) und (R2) erklärten Verknüpfungen sind von maximaler Genauigkeit in dem Sinne, daß zwischen dem in $I\!R$ ausgeführten Verknüpfungsergebnis $a \circ b$ und seiner Approximation $a \boxcirc b$ in R kein weiteres Element aus R liegt. Um dies einzusehen, nehmen wir an, es seien α und β benachbarte Elemente aus R mit der Eigenschaft

$$\alpha \leq a \circ b \leq \beta.$$

Anwendung von (R2), (R1) und (RG) auf diese Ungleichung liefert sofort, daß auch gilt

$$\alpha \leq a \boxcirc b \leq \beta,$$

d. h. $a \boxcirc b$ ist entweder gleich α oder gleich β.

Für besondere Anwendungen stellt PASCAL–XSC auch die gerichteten Rundungen $\bigtriangledown$ und $\bigtriangleup$ zur Verfügung, welche definiert sind durch die Forderungen (R1), (R2) und

$$\text{(R4)} \qquad \bigtriangledown a \leq a \text{ bzw. } a \leq \bigtriangleup a \qquad \text{für alle } a \in I\!R.$$

Diese Rundungen sowie die damit nach (RG) erklärten Verknüpfungen

$$a \mathbin{\bigtriangledown\!\!\!\!\!\circ}\; b := \bigtriangledown(a \circ b)$$

bzw.

$$a \mathbin{\bigtriangleup\!\!\!\!\!\circ}\; b := \bigtriangleup(a \circ b)$$

sind eindeutig bestimmt.

Neben den reellen Zahlen treten in der Numerik häufig auch Vektoren und Matrizen über den reellen Zahlen auf. Wir bezeichnen die betreffenden Mengen mit $VI\!R$ und $MI\!R$. Darüberhinaus verwendet man gelegentlich auch die komplexen Zahlen $\mathbb{C}$, Vektoren $V\mathbb{C}$ und Matrizen $M\mathbb{C}$ über den komplexen Zahlen. Alle diese Räume sind geordnet bezüglich der Ordnungsrelation $\leq$, welche in den Produkträumen komponentenweise erklärt wird. Mittels dieser Relation lassen sich Intervalle definieren. Numerische Algorithmen verwenden häufig auch Intervalle in den genannten Räumen. Wenn wir die Menge der Intervalle über einer geordneten Menge durch ein vorgestelltes I kennzeichnen, entstehen so die Räume $II\!R, IVI\!R, IMI\!R$ und $I\mathbb{C}, IV\mathbb{C}$ und $IM\mathbb{C}$. All diese Räume sind in der ersten Spalte der folgenden Tabelle noch einmal aufgelistet. Die auf einem Rechner darstellbaren Teilmengen der genannten Räume werden durch die in der zweiten Spalte der folgenden Tabelle aufgeführten Symbole beschrieben.

Grundräume der Numerik	auf dem Rechner darstellbare Teilmengen
$I\!R$	R
$VI\!R$	VR
$MI\!R$	MR
$II\!R$	IR
$IVI\!R$	VIR
$IMI\!R$	MIR
$\mathbb{C}$	CR
$V\mathbb{C}$	VCR
$M\mathbb{C}$	MCR
$I\mathbb{C}$	CIR
$IV\mathbb{C}$	$VCIR$
$IM\mathbb{C}$	$MCIR$

Die Arithmetik für einen Vektorrechner erklären wir nun durch alle in der zweiten Spalte der obigen Tabelle auftretenden inneren und äußeren Verknüpfungen. Wir fordern dabei, daß alle diese Verknüpfungen die Eigenschaften des Semimorphismus erfüllen. Da diese Vernüpfungen sich in den Produkträumen von den herkömmlicherweise auf einer Rechenanlage ausgeführten Verknüpfungen wesentlich unterscheiden, wollen wir ihre Definition hier noch einmal kurz wiederholen. Dazu sei S ein Element der linken Spalte der obigen Tabelle und T die in der Spalte rechts daneben stehende Teilmenge. Ferner sei

$$\square : S \to T$$

die Abbildung, welche die Elemente von S in diejenigen von T rundet. Von dieser Abbildung, der Rundung, verlangen wir wieder die Eigenschaften (R1) und (R2):

$$\text{(R1)} \quad \square a = a \quad \text{für alle } a \in T, \qquad \text{(Projektion)}$$

$$\text{(R2)} \quad a \leq b \Rightarrow \square a \leq \square b \quad \text{für alle } a, b \in S. \qquad \text{(Monotonie)}$$

Eine Rundung heißt antisymmetrisch, wenn zusätzlich gilt

$$\text{(R3)} \quad \square(-a) = -\square a \quad \text{für alle } a \in S. \qquad \text{(Antisymmetrie)}$$

Die Verknüpfungen in T werden wieder erklärt durch die Vorschrift

$$\text{(RG)} \quad a \boxdot b := \square(a \circ b) \quad \text{für alle } a, b \in T \text{ und } \circ \in \{+, -, \cdot, /\},$$

wobei $\circ$ für die mathematisch exakten Verknüpfungen in S steht.

Die dadurch in den Produkträumen erklärten Verknüpfungen (z. B. für komplexe Matrizen) sind wieder von höchster Genauigkeit in dem Sinne, daß komponentenweise zwischen dem in S ausgeführten Verknüpfungsergebnis $a \circ b$ und seiner Approximation $a \boxdot b$ in T kein weiteres Element aus T liegt.

Im Falle der in obiger Tabelle auftretenden Intervallräume steht anstelle der Ordnungsrelation $\leq$ in (R2) die Inklusion $\subseteq$. Von der Rundung $\square : IS \to IT$ fordern wir hier zusätzlich die Eigenschaft

(R4) $\quad a \subseteq \Box a \quad$ für alle $a \in IS$. (nach oben gerichtet)

Aus der Theorie (vgl. [20], [24]) ergibt sich, daß die Rundung damit eindeutig bestimmt ist.

Die herkömmliche Definition der Rechnerarithmetik unterscheidet sich ganz wesentlich von der oben angegebenen. Herkömmlicherweise versteht man unter Rechnerarithmetik nur die Verknüpfungen in R. Alle anderen Verknüpfungen in der zweiten Spalte der obigen Tabelle muß der Benutzer im Bedarfsfalle mittels bekannter Formeln selbst programmieren. Es geschieht dies üblicherweise in Form von Prozeduren. Jede in einem Algorithmus auftretende Verknüpfung verlangt dann einen eigenen Prozeduraufruf. Diese Vorgehensweise ist umständlich, zeitraubend aber vor allem viel zu ungenau. Betrachten wir etwa das Beispiel der Matrix-Multiplikation $C = A \cdot B$, welche komponentenweise die Ausführung eines Skalarproduktes erfordert. Dies geschieht üblicherweise auf der Basis der reellen Gleitkommaoperationen in der Form

$$C = (c_{ij}) = (a_{i1} \boxdot b_{1j} \boxplus a_{i2} \boxdot b_{2j} \boxplus \ldots \boxplus a_{in} \boxdot b_{nj}).$$

Im Gegensatz dazu verlangt die Formel (RG) eine Implementierung der Vorschrift $C = A \boxdot B$ mit

$$C = (c_{ij}) = (\Box(a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{in} \cdot b_{nj})).$$

Dabei treten in der inneren Klammer auf der rechten Seite die Addition und Multiplikation reeller Zahlen auf. Die Rundung darf nur am Schluß für jede Komponente ein einziges Mal ausgeführt werden. Der Rechner muß deshalb in der Lage sein, diese Formel für jedes beliebige n mit einer einzigen Rundung auszuführen. Dieses optimale Skalarprodukt spielt in allen Produkträumen der obigen Tabelle eine ausgezeichnete Rolle. Darüber hinaus wird das Skalarprodukt auch in numerischen Algorithmen häufig eingesetzt, um eine angestrebte hohe Ergebnisgenauigkeit zu erzielen.

Von einem Vektorrechner im mathematischen Sinne verlangen wir, daß er alle inneren und äußeren Verknüpfungen in den Räumen der rechten Spalte der obigen Tabelle bereitstellt. In PASCAL–XSC sind die Mengen in der rechten Spalte der obigen Tabelle vordefinierte Datentypen. Variable und Größen dieser Typen lassen sich mit den üblichen Operationszeichen `+`, `-`, `*`, `/` verknüpfen. In PASCAL–XSC bezeichnen diese Operationszeichen die oben verwendeten, mittels Semimorphismus definierten Operationen $\boxplus, \boxminus, \boxdot, \boxslash$. Auch Ausdrücke mit diesen Größen lassen sich damit einfach und übersichtlich niederschreiben. Alle inneren und äußeren Verknüpfungen erfüllen die an einen Semimorphismus gestellten Forderungen. Sofern die zugrundeliegende Hardware diese Forderungen unterstützt, führt dies im allgemeinen auf sehr schnelle Operationszeiten, welche im allgemeinen sogar schneller sind als wenn man die Verknüpfungen auf herkömmliche Weise mittels der Verknüpfungen in R programmiert und ausführt. Mangelt es an einer geeigneten Unterstützung durch die Hardware, so werden die semimorphen Verknüpfungen in den Räumen der zweiten Spalte der obigen Tabelle durch das Laufzeitsystem von PASCAL–XSC softwaremäßig über die gegebene *integer*-Arithmetik simuliert. Dies hat natürlich Laufzeiteinbußen zur Folge. Der Benutzer hat aber in jedem Falle eine wohldefinierte, umfangreiche Arithmetik zur Verfügung, auf deren Eigenschaften er in numerischen Algorithmen zuverlässig aufbauen kann. Von der arithmetischen und programmiersprachlichen Seite her gesehen, ist PASCAL–XSC damit eine ideale Vektorrechnersprache. Das Programmieren von Algorithmen im technisch-wissenschaftlichen Bereich wird durch die Spracherweiterung ganz wesentlich vereinfacht. PASCAL–XSC unterstützt die Entwicklung von Algorithmen mit hoher Genauigkeit und automatischer Ergebnisverifikation.

1.4 Einbettung in Programmiersprachen

Aus den Anforderungen an die Arithmetik ergeben sich die programmiersprachlichen Konzepte für eine Vektorsprache wie PASCAL–XSC oder ACRITH–XSC ziemlich zwangsläufig. Herkömmliche Programmiersprachen wie ALGOL, FORTRAN, PASCAL, MODULA oder PL/1 kennen als arithmetische Elementarverknüpfungen in der Regel nur die *integer*- und die *real*-Arithmetik. Alle anderen arithmetischen Verknüpfungen, insbesondere in den üblichen Produkträumen, müssen darauf zurückgeführt werden. Dies hat ein umständliches und vor allem zeitraubendes Hantieren mit höheren numerischen Einheiten wie Vektoren, Matrizen, komplexen Zahlen, komplexen Vektoren und Matrizen, sowie mit Intervallen über diesen Räumen zur Folge. Die einzelne Operation kann nur über einen Prozeduraufruf realisiert werden. Vor allem werden aber durch diese Vorgehensweise viele unnötige Rechenungenauigkeiten eingeschleppt.

PASCAL–XSC stellt demgegenüber alle Verknüpfungen in den oben genannten Räumen für vordefinierte Datentypen mit den üblichen Operationssymbolen zur Verfügung. Jede dieser Operationen ruft auf Maschinenniveau eine Elementaroperation auf, welche von höchster Genauigkeit und sehr effektiv implementiert ist. Die Operationen in den Produkträumen könnten dabei im allgemeinen sogar parallel ausgeführt werden. Im Unterschied zu den oben aufgeführten Sprachen stellt PASCAL–XSC insbesondere die folgenden Sprachelemente und Konstrukte bereit:

- Direkte Ansprechbarkeit der gerichteten Rundungen $\bigtriangledown$ und $\bigtriangleup$ und der dazu gehörigen Verknüpfungen $\overset{\circ}{\bigtriangledown}$ und $\overset{\circ}{\bigtriangleup}$ für alle $\circ \in \{+, -, \cdot, /\}$.
- Ein optimales Skalarprodukt für Vektoren beliebiger Länge.
- Intervalltypen mit zugehörigen Operatoren.
- Funktionen mit allgemeinem Ergebnistyp.
- Ein universelles Operatorkonzept.
- Überladen von Funktionsnamen und Operatoren.
- Dynamische, strukturierte numerische Datentypen.
- Eine große Anzahl von Standardfunktionen mit höchster Genauigkeit für die numerischen Grundtypen *real*, *complex*, *interval* und *complex interval*.

Darüber hinaus ist eine in PASCAL–XSC geschriebene Bibliothek von Problemlöseroutinen mit Ergebnissen von höchster Genauigkeit und automatischer Ergebnisverifikation (vgl. auch [23]) für eine große Anzahl von Standardproblemen der numerischen Mathematik verfügbar.

PASCAL–XSC ist damit eine echte Vektorsprache. Aufgrund der bereits in der Programmiersprache zum Ausdruck kommenden vektoriellen Notation der Verknüpfungen in den Produktmengen ist eine nachträgliche Vektorisierung von Programmen durch den Compiler häufig überflüssig. Die Ausführung dieser Operationen kann sowohl durch Parallelverarbeitung als auch durch Pipelinetechnik wesentlich beschleunigt werden. PASCAL–XSC wurde am Institut für Angewandte Mathematik der Universität Karlsruhe entwickelt und implementiert. ACRITH–XSC ist eine entsprechende FORTRAN 77 Erweiterung. Es wurde in Zusammenarbeit mit IBM ebenfalls am Institut für Angewandte Mathematik der Universität Karlsruhe entwickelt und implementiert. Inzwischen sind an verschiedenen Orten auch XSC-Erweiterungen von ADA, MODULA, APL und C als Compiler oder in Form von zusätzlichen Modulen realisiert worden.

1.5 Übersicht über PASCAL–XSC

Die Programmiersprache PASCAL–XSC wurde mit dem Ziel entwickelt, ein mächtiges Werkzeug für die numerische Lösung wissenschaftlicher Probleme zur Verfügung zu stellen, in dem die Rechnerarithmetik in den üblichen Räumen des numerischen Rechnens mathematisch exakt erklärt und implementiert ist (vgl. [20], [24]). Im wesentlichen umfaßt PASCAL–XSC die folgenden Konzepte

- Standard-PASCAL
- Universelles Operatorkonzept (benutzerdefinierte Operatoren)
- Funktionen und Operatoren mit beliebigem Ergebnistyp
- Überladen von Prozeduren, Funktionen und Operatoren
- Modulkonzept
- Dynamische Felder
- Zugriff auf Teilfelder
- Stringkonzept
- Kontrollierte Rundung
- Optimales (exaktes) Skalarprodukt
- Standarddatentyp *dotprecision* (Festkommaformat zur exakten Erfassung aller Gleitkomma-Produkte)
- Zusätzliche arithmetische Standardtypen wie *complex*, *interval*, *rvector* usw.
- Hochgenaue Arithmetik für alle Standarddatentypen
- Hochgenaue Standardfunktionen
- Exakte Auswertung von Ausdrücken (#-Ausdrücke)

Intervallarithmetik, komplexe Arithmetik, komplexe Intervallarithmetik sowie die entsprechenden Vektor- und Matrixarithmetiken werden in Form von Standard-Arithmetikmodulen bereitgestellt.

Darüber hinaus stehen in PASCAL–XSC geschriebene Anwendungsmodule zur Verfügung mit Routinen für häufig auftretende numerische Probleme, wie z. B.

- Lineare Gleichungssysteme
- Nichtlineare Gleichungssysteme
- Eigenwerte und Eigenvektoren
- Auswertung von arithmetischen Ausdrücken
- Optimierungsprobleme
- Auswertung von Polynomen und Nullstellenbestimmung
- Numerische Quadratur
- Anfangs- und Randwertprobleme von gewöhnlichen Differentialgleichungen
- Integralgleichungen
- Automatische Differentiation

Alle diese Problemlöseroutinen liefern **automatisch verifizierte Ergebnisse**.

Die wichtigsten der neuen Konzepte sollen nun kurz erläutert werden.

1.5.1 Universelles Operatorkonzept und allgemeiner Ergebnistyp

PASCAL–XSC erleichtert die Programmierung erheblich durch die Möglichkeit, Funktionen und auch Operatoren mit allgemeinem Ergebnistyp zu vereinbaren. An dem einfachen Beispiel der Polynomaddition werden die Vorteile, die diese Konzepte mit sich bringen, deutlich.
Hat man in einem Programm den Typ `polynom` vereinbart gemäß

```
const n = 20;
type  polynom = array [0..n] of real;
```

so muß man in Standard-PASCAL die Addition zweier Polynome als *Prozedur*

```
procedure add (a,b: polynom; var c: polynom);
  var i: integer;
  begin
    for i:= 0 to n do
      c[i]:= a[i] + b[i];
  end;
```

implementieren und für die Berechnung des Ausdrucks $z = a + b + c + d$ mehrere Aufrufe von `add` verwenden:

```
add (a,b,z);
add (z,c,z);
add (z,d,z);
```

In PASCAL–XSC hingegen ist es aufgrund des allgemeinen Ergebnistyps möglich, eine *Funktion*

```
function add (a,b: polynom): polynom;
  var i: integer;
  begin
    for i:= 0 to n do
      add[i]:= a[i] + b[i];
  end;
```

zu verwenden, mit deren Hilfe sich der Ausdruck $z = a + b + c + d$ dann schon etwas einfacher durch `z:= add(a,add(b,add(c,d)))` berechnen läßt. Implementiert man dagegen in PASCAL–XSC den *Operator*

```
operator + (a,b: polynom) resp : polynom;
  var i: integer;
  begin
    for i:= 0 to n do
      resp[i]:= a[i] + b[i];
  end;
```

so läßt sich die ursprüngliche Formel in der gewohnten mathematischen Notation `z:= a+b+c+d` programmieren.

Neben der Möglichkeit, Operatorsymbole zu verwenden, kann man in PASCAL–XSC auch Operatoren mit Namen definieren, muß diesen dann aber in einer vorangestellten Prioritätsvereinbarung eine Priorität zuweisen.

1.5.2 Überladen von Prozeduren, Funktionen und Operatoren

PASCAL–XSC erlaubt das Überladen von Funktions- und Prozedurnamen. Dadurch wird ein generisches Namenskonzept in die Sprache eingeführt, das es dem Benutzer z. B. erlaubt, die Bezeichner *sin, cos, exp, ln, arctan* und *sqrt* außer für *real*-Zahlen auch für Intervalle, komplexe Zahlen und Elemente anderer mathematischer Räume zu verwenden. Dabei werden Funktions- und Prozedurnamen durch Anzahl, Reihenfolge und Typ der Parameter unterschieden. Der Ergebnistyp wird *nicht* zur Unterscheidung herangezogen.

Ebenso können, wie bereits weiter oben gezeigt, Operatoren überladen werden, sodaß die üblichen Standardoperatorzeichen auch für beliebige, selbstdefinierte Typen verwendet werden können. Damit bietet PASCAL–XSC eine wesentliche Erweiterung bzw. Verallgemeinerung des Ausdruckskonzepts.

Durch die ebenfalls mögliche Überladung des Zuweisungsoperators := können bei Wertzuweisungen mathematische Schreibweisen verwendet werden:

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

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

...

  r:= 1.5;
  c:= r;  {Komplexe Zahl mit Realteil 1.5 und Imaginaerteil 0}
```

1.5.3 Modulkonzept

Das Modulkonzept ermöglicht es, große Programme in Module aufzuteilen und diese dann getrennt zu entwickeln und zu übersetzen. Die Kontrolle der Syntax und der Semantik kann über die Modulgrenzen hinweg durchgeführt werden. Module werden eingeleitet mit dem Wortsymbol **module**, gefolgt von einem Namen und einem Strichpunkt. Der Rumpf ist wie bei einem herkömmlichen PASCAL-Programm aufgebaut, mit der Ausnahme, daß zur Kennzeichnung der exportierten Objekte des Moduls das Wortsymbol **global** vor den Wortsymbolen **const**, **type**, **var**, **procedure**, **function** und **operator** sowie unmittelbar nach **use** und dem Gleichheitszeichen bei Typdeklarationen stehen darf. Damit ist es möglich, anonyme und nicht anonyme Typen zu vereinbaren.

Von anderen Modulen oder Programmen *importiert* werden Module mit der **use**-Anweisung, die bewirkt, daß sämtliche im importierten Modul mit **global** vereinbarten Objekte auch im importierenden Modul oder Programm bekannt sind.

Am Beispiel eines Polynomarithmetik-Moduls wird der schematische Aufbau eines Moduls demonstriert:

```
module poly;
  use { andere Module }
  ...
  { lokale Vereinbarungen }
  ...
  { globale Vereinbarungen }
  global type  polynom = ...
  ...
  global procedure read (...
  ...
  global procedure write (...
  ...
  global operator + (...
  ...
```

```
  global operator * (...
  ...
begin
  { Initialisierungsteil des Moduls}
  ...
end. {module poly}
```

1.5.4 Dynamische Felder, Teilfelder

Das Konzept der dynamischen Felder ermöglicht es, Algorithmen unabhängig von der Größe der verwendeten Felder zu implementieren. Die Indexbereiche von dynamischen Feldern werden erst zur Laufzeit und nicht bereits zur Übersetzungszeit festgelegt. In PASCAL–XSC können Unterprogramme voll dynamisch realisiert werden, da die Allokierung und Freigabe von lokalen dynamischen Variablen automatisch erfolgt. Dadurch wird der Speicherplatz optimal ausgenutzt.

Ein dynamischer Typ `polynom` kann z. B. in folgender Form vereinbart werden

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

Bei der Vereinbarung von Variablen dieses dynamischen Typs müssen dann die Indexgrenzen spezifiziert werden:

```
var  p, q : polynom [0..2*n];
```

Um auf die erst zur Laufzeit bekannten Grenzen dynamischer Felder zugreifen zu können, stehen die beiden Funktionen *lbound(...)* und *ubound(...)* bzw. deren Abkürzungen *lb(...)* und *ub(...)* zur Verfügung. Eine Multiplikation zweier Polynome kann dynamisch beispielsweise wie folgt realisiert werden:

```
operator * (a,b:polynom) rm: polynom[0..ub(a)+ub(b)];
  var i,j: integer;
      r  : polynom[0..ub(a)+ub(b)];
  begin
    for i:= 0 to ub(a)+ub(b) do
        r[i]:= 0;
    for i:= 0 to ub(a) do
      for j:= 0 to ub(b) do
        r[i+j]:= r[i+j] + a[i] * b[j];
    rm:= r;
  end;
```

Ein in PASCAL–XSC geschriebenes Programm, das dynamische Felder verwendet, sollte schematisch wie folgt aufgebaut sein:

```
program dynamik (input,output);
...
type polynom = dynamic array [*] of real;
...
var  n : integer;
```

```
...
operator * (a,b:polynom)...
...
procedure write (var f : text; p: polynom);
...
procedure main (n : integer);
  var
    p,q,s : polynom[0..n];
    r     : polynom[0..2*n];
  begin
    ...
    r:= p * q;
    writeln ('p*q = ', r);
    ...
  end;

begin {Hauptprogramm}
  read (n);
  main (n);
end.
```

Man kann sowohl auf Zeilen als auch auf Spalten dynamischer Felder zugreifen, wie das folgende Beispiel zeigt:

```
type vector =  dynamic array [*] of real;
type matrix =  dynamic array [*] of vector;
var  v : vector[1..n];
     m : matrix[1..n,1..n];
...
  v      := m[i];   { i-te Zeile  von m }
  m[*,j] := v;      { j-te Spalte von m }
```

1.5.5 Stringkonzept

In die Sprache PASCAL–XSC wurde ein Stringkonzept integriert, das die Handhabung von Zeichenketten variabler Längen ermöglicht. Vereinbarung, Eingabe und Ausgabe von Strings sind gegenüber Standard PASCAL wesentlich vereinfacht, die Anwendung von speziellen Standardfunktionen und Operatoren erlaubt die Formulierung von Stringausdrücken und bietet vielfache Möglichkeiten für Stringmanipulationen.

1.5.6 Arithmetik und Rundung

Die Menge der Standardoperatoren für *real*-Zahlen ist in PASCAL–XSC gegenüber Standard-PASCAL um die gerichtet rundenden Operatoren $\circ$< und $\circ$> mit $\circ \in \{+, -, *, /\}$ erweitert. Dadurch werden die Verknüpfungen mit gerichteter Rundung nach unten und oben beschrieben.

In den Arithmetikmodulen werden die üblichen Operatoren auch für komplexe Zahlen, Intervalle und komplexe Intervalle wie auch für Vektoren und Matrizen über diesen Räumen zur Verfügung gestellt.

1.5.7 Exakte Auswertung von Ausdrücken

Für die Implementierung von Einschließungsalgorithmen mit hoher Genauigkeit benötigt man die exakte Auswertung von Skalarprodukten. Um dieses zu ermöglichen, wurde in PASCAL–XSC der neue Datentyp *dotprecision* eingeführt, der ein Festkommaformat darstellt, in dem skalare Ergebnisse, insbesondere Summen von Gleitkommaprodukten, exakt gespeichert werden können.

Weiterhin können Skalarproduktausdrücke in Vektor- und Matrixform mit nur einer einzigen Rundung pro Komponente durch exakte Auswertung von Ausdrücken in Form sogenannter Lattenkreuzausdrücke (#-Ausdrücke) berechnet werden.

Kapitel 2

Sprachbeschreibung

PASCAL–XSC basiert auf der wohlbekannten Programmiersprache PASCAL, die in dem Bericht von Jensen und Wirth [11] beschrieben ist. Da PASCAL–XSC eine Erweiterung von PASCAL ist, geben wir keine detaillierte Beschreibung der gesamten Sprache (siehe dazu z. B. [8], [11] oder [12]), sondern nur eine kurze Beschreibung der Standardelemente von PASCAL und eine etwas gründlichere Einführung in die neuen Sprachelemente.

Die Syntax der Sprache wird in einer unmittelbar lesbaren, vereinfachten Backus-Naur-Form angegeben und jeweils am Rand des Textes durch einen schwarzen Balken gekennzeichnet. Dabei werden die Grundsymbole in der in Abschnitt 2.1 verwendeten typographischen Darstellung wiedergegeben, Sprachelemente (Syntaxvariable) sind Hauptworte der Umgangssprache. Beliebige Wiederholungen von Teilen der Syntax werden durch die Zeichenfolge ... gekennzeichnet. Diese Kennzeichnung bezieht sich immer auf alle in der Zeile vorausgehenden Sprachelemente, wobei diese auch ganz fehlen dürfen, wenn kein anderslautender Kommentar folgt.

Beispiel:

> **var**
>   Namensliste : Typ; ... { nicht leer }

Wie im Beispiel treten an verschiedenen Stellen der Syntax Listen auf (hier Namensliste), welche stets aus einer nichtleeren Folge von entsprechenden Objekten bestehen, wobei die Listenelemente durch Kommata getrennt sind.

Beispiel 2.0.1:

Die oben gegebene Syntax für Variablenvereinbarung erlaubt etwa folgenden Programmtext:

```
var i, j, k : integer;
    x, y    : real;
    m       : array [1..10, 1..10] of real;
```

Die über Standard-PASCAL hinausgehenden neuen Konstrukte erscheinen optisch gut sichtbar in folgendem Rahmen:

PASCAL–XSC

Innerhalb dieses Rahmens stehen grundsätzlich alle für PASCAL–XSC gültigen Regelungen, die über Standard-PASCAL hinausgehen!

Einen kompakten Überblick über die Syntax der Sprache PASCAL–XSC geben die *Syntaxdiagramme* im Anhang A ab Seite 271.

Für alle in diesem und den folgenden Kapiteln erwähnten Implementierungsabhängigkeiten wird auf das mit der jeweiligen Compilerversion ausgelieferte Benutzerhandbuch verwiesen.

2.1 Grundsymbole

Zur Formulierung eines Programmes dienen ausschließlich die folgenden Grundsymbole:

Buchstaben:	a, b, c, . . ., z							
Ziffern:	0, 1, 2, . . ., 9							
Sonderzeichen:	<=	<	>	>=	=	<>		
	(	)	[	]	{	}		
	+	−	*	/				
	:=	.	,	;	:	'	↑	⊔	..

Dabei bedeutet ⊔ das Leerzeichen.

Alternativ für die Zeichen	{	}	[	]	↑
können die Zeichen	(*	*)	(.	.)	@ oder ^

verwendet werden.

Wortsymbole: **and**, **array**, **begin**, **case**, **const**, **div**, **do**, **downto**, **else**, **end**, **file**, **for**, **forward**, **function**, **goto**, **if**, **in**, **label**, **mod**, **nil**, **not**, **of**, **or**, **packed**, **procedure**, **program**, **record**, **repeat**, **set**, **then**, **to**, **type**, **until**, **var**, **while**, **with**

Buchstaben können in Groß- und Kleinschreibweise dargestellt werden. Eine Unterscheidung zwischen Groß- und Kleinbuchstaben wird jedoch nicht vorgenommen. So sind z. B. die Namen *PASCAL* und *Pascal* identisch. Ein Wortsymbol kann beliebig in Großbuchstaben und Kleinbuchstaben geschrieben werden.

PASCAL–XSC

Zusätzliche Grundsymbole

Buchstaben:	_ (Unterstrich)			
Sonderzeichen:	$	#		
	#*	#<	#>	##
	><	+*	**	
	+>	−>	*>	/>
	+<	−<	*<	/<
Wortsymbole:	**dynamic**,	**external**,	**global**,	**module**,
	operator,	**priority**,	**sum**,	**use**

2.2 Bezeichner (Namen)

Zur Bezeichnung der in einer Programmiersprache auftretenden Objekte wie Konstanten, Variablen, Typen, Funktionen, usw. werden Bezeichner verwendet.

Ein Bezeichner besteht aus einer beliebigen Folge von Buchstaben und Ziffern, beginnend mit einem Buchstaben.

Beispiel 2.2.1:

variable1, nordwest, extrem, ab

Zwei Bezeichner sind identisch, wenn sie beide aus derselben Folge von Symbolen bestehen.

Wortsymbole sind als Bezeichner nicht zulässig! Bei der Wahl eines Bezeichners ist zu beachten, daß folgende Bezeichner (Standardnamen) eine vordefinierte Bedeutung haben:

abs	eof	ln	pred	round	trunc
arctan	eoln	maxint	put	sin	write
boolean	exp	new	read	sqr	writeln
char	false	odd	readln	sqrt	
chr	get	ord	real	succ	
cos	input	output	reset	text	
dispose	integer	page	rewrite	true	

Diese Bezeichner können ohne explizite Vereinbarung in dieser Bedeutung verwendet werden. Werden sie in einer expliziten Vereinbarung neu festgelegt, so können sie nur mit dieser neuen Bedeutung verwendet werden.

PASCAL–XSC

Der Unterstrich _ (underscore) darf an beliebiger Stelle eines Bezeichners auftreten.

Beispiel 2.2.2:

variable_1, nord_west, _extrem_, a__b, _, __

Maximale Länge eines Bezeichners ist die logische Länge einer Zeile.

Zur Identifizierung von Namen wird zwischen Groß- und Kleinbuchstaben nicht unterschieden. Demnach bedeuten *nord_west* und *Nord_West* denselben Namen.

PASCAL–XSC

Weitere vordefinierte Bezeichner sind:

arccos	cmatrix	im	log2	setlength
arccot	comp	image	log10	sign
arcsin	complex	imatrix	mant	sinh
arcosh	cosh	inf	mark	string
arcoth	cot	interval	maxlangth	substring
arctan2	coth	ivector	pos	sup
arsinh	cvector	ival	re	tan
artanh	dotprecision	lb	release	tanh
cimatrix	expo	lbound	rmatrix	ub
cinterval	exp2	length	rvector	ubound
civector	exp10	loc	rval	

Die Standardnamen von Prozeduren und Funktionen können überladen werden, sodaß sie sowohl in ihrer ursprünglichen Bedeutung als auch in der neuen Bedeutung verwendet werden können (vgl. auch Abschnitt 2.7.10).

Bei Benutzung eines Moduls mittels der **use**-Klausel werden alle dort mit **global** vereinbarten Bezeichner zu vordefinierten Bezeichnern im benutzenden Modul oder Programm (vgl. auch Abschnitt 2.8).

Namen werden auch zur Bezeichnung von Operatoren verwendet (vgl. auch Abschnitt 2.7.6).

2.3 Konstanten, Typen und Variablen

An Standarddatentypen bietet PASCAL wie üblich die Wertebereiche *integer*, *real*, *boolean* und *char* mit den entsprechenden Operatoren (vgl. Abschnitt 2.4). Daneben kann durch Aufzählung der zugehörigen Werte ein Aufzählungstyp vereinbart werden. Die Werte eines solchen einfachen Datentyps bezeichnet man als Literalkonstante. Die Schreibweise dieser Literalkonstanten ist fest vorgeschrieben (siehe Abschnitt 2.3.1). Zu den strukturierten Datentypen Array, Set, Record und File gibt es keine Literalkonstanten, ausgenommen die Stringkonstanten im Falle des eindimensionalen Arrays mit Komponententyp *char*. Dabei handelt es sich um Zeichenketten, die in Apostrophe eingeschlossen sind.

In einer Konstantendefinition können Namen für Konstanten (benamte Konstanten) festgelegt werden:

```
const
   Name = Konstante; ... { nicht leer }
```

Die rechts stehende Konstante ist dabei eine Literalkonstante oder eine bereits zuvor definierte Konstante.

Beispiel 2.3.1:

```
const
   n     = 50;
   eps   = 10e−13;
   k     = n;
   zf    = 'zeichenfolge';
```

Benamte Konstanten können wie Literalkonstanten in einem Programm verwendet werden. Sie sind während der Programmausführung unveränderlich.

In einer Typdefinition können Namen für Typen festgelegt werden:

```
type
   Name = Typ; ... { nicht leer }
```

Der rechts stehende Typ ist dabei ein explizit angegebener Typ oder ein zuvor definierter Typ. Die eingangs erwähnten Standardtypen gelten als vordefiniert.

Beispiel 2.3.2:

```
type farbe    =  (rot, blau, gelb);
     logisch  =  boolean;
     vektor   =  array [1..20] of real;
```

In einer Variablenvereinbarung können Namen für Variablen festgelegt werden:

> **var**
> Namensliste: Typ; ... { nicht leer }

Die in der Liste aufgezählten Namen bezeichnen Variablen zum rechts stehenden Typ. Eine Variable kann als symbolische Adresse eines entsprechenden Speicherplatzes interpretiert werden.

Beispiel 2.3.3:

var	i, j, k	:	integer;
	x, y	:	real;
	f	:	farbe;
	vek1, vek2	:	vektor;
	m	:	**array** [1..20] **of** vektor;

PASCAL–XSC

Als weitere Standarddatentypen stehen *dotprecision, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix* und *string* zur Verfügung.

Bei Verwendung eines dynamischen Feldtyps in einer Variablenvereinbarung müssen die Indexgrenzen durch entsprechende Ausdrücke spezifiziert sein (siehe Abschnitt 2.3.2 über dynamische Feldtypen).

2.3.1 Einfache Datentypen

Zu den einfachen Datentypen zählen die Standardtypen *integer*, *real*, *boolean* und *char*. Hinzu kommen die Aufzählungstypen (auch Code-Typen genannt) und die Unterbereichstypen. Diese sind im einzelnen in folgender Weise festgelegt:

integer Implementierungsabhängige Teilmenge der ganzen Zahlen. Die vordefinierte Konstante *maxint* bezeichnet die implementierungsabhängig größte ganze Zahl. Ein Literalkonstante des Typs *integer* ist eine Ziffernfolge (von Dezimalziffern) mit oder ohne Vorzeichen + oder −.

Beispiel 2.3.4:

128 −30 +4728 007

real Implementierungsabhängige Teilmenge der reellen Zahlen. Eine Literalkonstante des Typs *real* hat die Darstellung

± Mantisse E Exponent

Dabei ist die Mantisse eine Ziffernfolge mit oder ohne Dezimalpunkt, der Exponent ein *integer* Wert (implementierungsabhängig beschränkt). Es ist auch die Darstellung

$\pm$ Mantisse

ohne Exponententeil erlaubt. Grundsätzlich muß vor und nach dem Dezimalpunkt mindestens eine Ziffer stehen.

Beispiel 2.3.5:

$$\begin{array}{r} 3.1726\mathrm{E}{-2} \\ -0.008\mathrm{E}{+5} \\ +1\mathrm{E}{-10} \\ 3.1415 \end{array}$$

Es ist zu beachten, daß der Wert der dezimalen Zahlendarstellung von PASCAL nicht immer in der implementierungsabhängigen Menge des Typs *real* enthalten sein muß. So ist etwa die dezimale Gleitpunktzahl 1.1 nicht exakt als duale Gleitpunktzahl darstellbar. Die in einem Programm verwendete Literalkonstante 1.1 stellt in diesem Fall einen *real*-Wert dar, der nicht dem reellen Wert 1.1 entspricht. Diese Problematik der Konvertierung ist grundsätzlich zu beachten, wenn Literalkonstanten in Ausdrücken als Operanden oder in Funktionen und Prozeduren als Argumente auftreten oder einzulesen sind.

boolean — Der Wertebereich besteht aus den beiden logischen Konstanten *true* und *false*. Es gilt *false* < *true*.

char — Der Wertebereich ist eine von der jeweiligen Implementierung abhängige Menge von Zeichen. Literalkonstanten werden in Apostrophe eingeschlossen. Es gilt

'0' < '1' < ... < '9'
und
'a' < 'b' < ... < 'z'.

Aufzählungstypen — Der Wertebereich besteht aus den in der Typdefinition aufgezählten Konstanten (geordnete Folge von Namen). Die Ordnung ist durch die Reihenfolge der Aufzählung bestimmt. Ein Aufzählungstyp wird in einer Typdefinition vom Programmierer festgelegt und darf in keinem Wert mit einem anderen Aufzählungstyp kollidieren.

Beispiel 2.3.6:

Die Typdefinition

type farbe = (rot, blau, gelb);

definiert den Aufzählungstyp *farbe* mit den Werten *rot, blau* und *gelb*. Ein weiterer Typ

sonderfarbe = (gelb, orange);

ist nicht zulässig, da der Wert *gelb* bereits im Typ *farbe* auftritt.

Unterbereichstypen

Zu den Standarddatentypen *integer, boolean, char* sowie zu allen Aufzählungstypen (Grundbereiche) können Unterbereichstypen definiert werden durch Angabe von unterer und oberer Grenze in der Form

Konstante .. Konstante

Der Wertebereich eines Unterbereiches besteht aus der unteren und oberen Grenze sowie allen dazwischen liegenden Werten des Grundbereiches. Dabei wird die Ordnung des Grundbereiches übernommen. Die untere Grenze muß kleiner oder gleich der oberen Grenze, beide müssen vom gleichen Grundbereichstyp sein.

Beispiel 2.3.7:

type

ub	=	1..100;
teilfarbe	=	blau..gelb;
buchstaben	=	'a'..'z';
oktalziffern	=	0..7;

PASCAL–XSC

integer

Ein Wert des Typs *integer* kann auch als hexadezimale Konstante geschrieben werden, die aus einer durch das Symbol $ eingeleiteten hexadezimalen Ziffernfolge aus den Ziffern $0, 1, \ldots, 9$ und A,B,...,F bzw. a,b,...,f besteht.

Beispiel 2.3.8:

$12AFB2

PASCAL–XSC

real Um die unvermeidbare Konvertierung von Literalkonstanten in das interne (endliche) Datenformat für *real*-Zahlen kontrolliert durchführen zu können, wird eine zusätzliche Notation für reelle Literalkonstanten notwendig. Während die übliche PASCAL-Darstellung von *real*-Zahlen die Konvertierung mit Rundung zur nächstgelegenen Maschinenzahl impliziert, können durch die Schreibweisen

(< ± Mantisse E Exponent)

bzw.

(> ± Mantisse E Exponent)

real-Konstanten spezifiziert werden, die bei der Konvertierung zur nächstkleineren bzw. zur nächstgrößeren Maschinenzahl gerundet werden. Dabei kann das E und der Exponententeil wie üblich auch entfallen.

Beispiel 2.3.9:

(< 1.1)
(> −1.0E−1)

dotprecision Der Datentyp *dotprecision* baut auf dem Datentyp *real* auf und erlaubt die Darstellung von Produkten zweier beliebiger *real*-Zahlen und die exakte Summation beliebig vieler solcher Produkte in einem Festkommaformat geeigneter Größe. Ist das interne *real*-Format durch die Mantissenlänge l und den minimalen bzw. maximalen Exponenten *emin* bzw. *emax* festgelegt (vgl. Abschnitt 2.4.1.2), so belegt eine *dotprecision*-Variable einen Speicherbereich der Form

$$\underbrace{\boxed{\qquad\quad}}_{g}\underbrace{\boxed{\qquad\qquad\qquad}}_{2\cdot\text{emax}}\underbrace{\boxed{\quad}}_{2\cdot l}\underbrace{\boxed{\qquad\qquad\qquad}}_{2\cdot|\text{emin}|}$$

mit der Gesamtlänge $L = g + 2emax + 2|emin| + 2l$ Ziffern. Dabei bezeichnet g die implementierungsabhängige Anzahl von Schutzziffern, die für das Auffangen von Überläufen bei der Summation notwendig sind (vgl. dazu [24] und [25]).

PASCAL–XSC

Typischerweise treten Werte vom Typ *dotprecision* bei der Multiplikation von Vektoren oder Matrizen auf (Skalarprodukte). Diese Werte können in dem Format dieses Typs exakt, d. h. ohne Rundungsfehler, dargestellt werden, unabhängig von der Größe der Vektoren bzw. Matrizen.

Werte des Typs *dotprecision* können ausschließlich über #-Ausdrücke (vgl. Abschnitt 2.4.2) in der Form

(Genau Auszuwertender Ausdruck)

erzeugt werden, d. h. es gibt keine (Literal-) Konstante zu diesem Typ.

2.3.2 Strukturierte Datentypen

In PASCAL gibt es bekanntlich vier verschiedene Strukturierungsarten:

- Felder (Arrays)
- Dateien (Files)
- Verbunde (Records)
- Mengen (Sets)

Sie unterscheiden sich in der Art und Weise, wie man Elemente der Standardtypen zu einer höheren Struktur zusammenfassen und auf deren Komponenten zugreifen kann. Die volle Komplexität wird dadurch erreicht, daß auch beliebige Strukturen selbst wieder als Komponenten einer Datenstruktur auftreten dürfen.

Jede Typdefinition darf mit dem Wortsymbol **packed** beginnen, wodurch implementierungsabhängig eine platzsparende Speicherung der Werte des Komponententyps bewirkt werden kann. Für die Semantik hat **packed** keine Wirkung.

2.3.2.1 Felder (Arrays)

Ein Feld, auch Array genannt, besteht aus einer zu vereinbarenden festen Anzahl von Komponenten gleichen Typs. Die einzelnen Komponenten werden durch Indizes gekennzeichnet, die als Werte von sogenannten Indexausdrücken berechenbar sind. Die Typdefinition eines Feldes muß also die Indextypen und den Komponententyp enthalten:

array [Indextypliste] **of** Komponententyp

Dabei ist ein Indextyp ein Unterbereichstyp von *integer, boolean, char* oder eines Aufzählungstyps oder einer der drei letzten Typen selbst. Der Komponententyp darf ein beliebiger Typ sein. Man beachte jedoch den Speicheraufwand bei Verwendung des Komponententyps *dotprecision*.

Beispiel 2.3.10:

array [1..10] **of** real	{ Vektor mit 10 reellen Komponenten }
array [1..10, 1..10] **of** real	{ reelle Matrix mit 10 Zeilen und 10 Spalten }
array ['a'..'z'] **of** boolean	{ Vektor mit 26 (logischen) Komponenten }
array [1..10] **of** **array** ['a'..'z'] **of** boolean	{ logische Matrix mit 10 Zeilen und 26 Spalten }
array [(rot, gelb, blau, schwarz)] **of** 0..10	{ Vektor mit 4 Komponenten vom Unterbereichstyp 0..10 }

Die Komponenten eines Feldes können als Variable (Komponentenvariable) verwendet werden, wobei der Zugriff folgende Form hat:

Feldname [Indexausdrucksliste]

oder

Feldname [Indexausdrucksliste]
[Indexausdrucksliste] ...

Dabei ist zu beachten, daß die Zuordnung zwischen Indexausdrücken (Indizes) und Indexbereichen von links nach rechts geschieht und die Indizes innerhalb des zugehörigen Indexbereiches liegen müssen.

Beispiel 2.3.11:

Vereinbarung	Komponentenvariablen
var v: **array** [1..10] **of** real;	v[1], …, v[10]
var m: **array** [1..10,1..10] **of** real;	m[1,1], …, m[1,10], m[2,1], …, m[2,10], ⋮ m[10,1], …, m[10,10] möglich ist auch m[1][1], …
var feld: **array** [1..10] **of** **array** ['a'..'z'] **of** boolean;	feld[1]['a'], …, feld[1]['z'], ⋮ feld[10]['a'],…, feld[10]['z'] möglich ist auch feld[1,'a'], …

2.3.2.2 Teilfelder

Ist beim Zugriff auf eine Komponentenvariable die Anzahl k der angegebenen Indizes kleiner als die vereinbarte Anzahl n von Indexbereichen (n-dimensionales Feld), so ist damit ein $n-k$-dimensionales Teilfeld (Komponentenvariable) angesprochen. Die angegebenen k Indizes beziehen sich auf die ersten k vereinbarten Indexbereiche. Kann der Komponentenvariablen aufgrund der Typdefinition oder der Variablenvereinbarung kein expliziter Typ zugeordnet werden, so spricht man von einer Variablen von *anonymem Typ* (vgl. auch Abschnitt 2.3.5 zur Verträglichkeit von Typen).

Beispiel 2.3.12:

Ist die Variable m vereinbart durch

var m: **array** [1..10, 1..20] **of** real;

dann bezeichnet die Komponentenvariable $m[5]$ ein eindimensionales Teilfeld von anonymem Typ, und zwar einen Vektor mit 20 Komponenten, bestehend aus der 5. Zeile der Matrix m.

PASCAL–XSC

Man kann beliebige Teilfelder (Komponentenvariablen) eines Feldes dadurch ansprechen, daß man gewisse auszuwählende Indexbereiche offen läßt. Dies wird mit dem Zeichen $*$ in der Indexausdruckliste gekennzeichnet. Wenn einem $*$ kein weiterer Indexausdruck mehr folgt, kann er auch entfallen.

Beispiel 2.3.13:

Entsprechend einer Vereinbarung

var m: **array** [1..10, 1..20] **of** real;

bezeichnet die Komponentenvariable $m[*,1]$ eine Feldvariable, und zwar einen Vektor mit 10 Komponenten, bestehend aus der 1. Spalte der Matrix m.

Die Schreibweisen

m[1,*]	und	m[1]		bzw.
m	und	m[*]	und	m[*,*]

sind äquivalent.

2.3.2.3 Zugriff auf Indexgrenzen

PASCAL–XSC

Zum Ansprechen von Indexgrenzen unabhängig von den aktuellen verwendeten Größen bei der Feldvereinbarung (unbedingt notwendig bei der Verwendung dynamischer Felder) sind die zwei Standardfunktionen

ubound (Feldvariable, I Konstante) und
lbound (Feldvariable, I Konstante)

vorgesehen, die als Wert die obere (upper) und untere (lower) Schranke des i-ten Indexbereiches (i = Wert der I Konstante) der Feldvariablen liefern. Die I Konstante kann auch fehlen. In diesem Fall wird implizit der erste Indexbereich angesprochen. Das Ansprechen eines nicht existierenden Indexbereichs ist nicht zulässig.

Abkürzend können auch die Schreibweisen *lb* (für *lbound*) und *ub* (für *ubound*) verwendet werden.

Beispiel 2.3.14:

```
type matrix = array [1..n,1..k] of real;
function summe (var m: matrix): real;
var
   i, j: integer;
   s: real:
begin
   s := 0;
   for i := lbound(m) to ubound(m) do
      for j:= lb(m,2) to ub(m,2) do
         s := s + m[i,j]:
   summe := s
end;
```

2.3.2.4 Dynamische Felder

Standard-PASCAL läßt dynamische Felder nicht zu. Eine gewisse Dynamik findet sich in der Stufe 1 des Standards (vgl. [8]) in der Spezifikation von Feldargumenten bei Funktionen und Prozeduren durch die *Konform-Array-Schemata* (siehe Abschnitt 2.7.1). Diese erlauben bei Aufrufen der entsprechenden Prozeduren und Funktionen aktuelle Felder, die nicht von vornherein von einem bestimmten Array-Typ sein müssen.

PASCAL–XSC

Wie auch andere bekannte Programmiersprachen (z. B. ALGOL 60, ALGOL 68 [35], ADA [13]) sieht PASCAL–XSC die Möglichkeit der *dynamischen Feldvereinbarung* vor. Dies bedeutet im wesentlichen, daß Feldvariablen innerhalb von Unterprogrammen nicht grundsätzlich statisch wie in Standard-PASCAL zu vereinbaren sind, sondern daß in den Indexgrenzen Ausdrücke auftreten können, die bei jedem Unterprogrammaufruf zu neuen Indexgrenzen führen können. Die Einführung dynamischer Feldtypen und deren Verwendung bei der Spezifikation von formalen Argumenten enthält insbesondere die gesamte Funktionalität der Konform-Array-Schemata von Standard-PASCAL, verwendet jedoch eine etwas andere Syntax.

Zunächst können wie bei statischen Feldern auch für dynamische Felder Typdefinitionen vorgenommen werden durch

dynamic array [Dimensionsliste] **of** Komponententyp

In der Dimensionsliste wird dabei jeder Indexbereich durch ein Zeichen $*$ markiert.

Ein dynamischer Feldtyp darf nicht Komponententyp eines strukturierten Typs sein, ausgenommen eines dynamischen Feldtyps selbst. Man beachte, daß obige Typdefinition nur die Anzahl der Indexbereiche des Feldes und den Komponententyp festlegt.

Beispiel 2.3.15:

type dynpolynom = **dynamic array** [*] **of** real;
dynvektor = **dynamic array** [*] **of** real;
dynmatrix = **dynamic array** [*,*] **of** real;

Nicht zulässig ist z. B.:

type falschertyp = **dynamic array** [1..n,*] **of** real;

Dynamische Feldtypen dürfen in der Variablenvereinbarung unter Angabe (Spezifikation) entsprechender Indexausdrücke auftreten. Hierbei kann entweder ein vorher vereinbarter Typname oder ein explizit geschriebener dynamischer Typ verwendet werden.

Beispiel 2.3.16:

var mat1: dynmatrix [1..n,1..2*n];
mat2: **dynamic array** [1..n,1..2*n] **of** real;

PASCAL–XSC

In beiden Fällen muß die Berechnung der jeweiligen Indexausdrücke zum Zeitpunkt der Verarbeitung der Variablenvereinbarung gewährleistet sein. Echte Dynamik bei der Feldvereinbarung kann also nur innerhalb von Prozeduren und Funktionen unter Verwendung globaler Größen oder formaler Argumente in den Indexausdrücken auftreten (siehe dazu Abschnitt 2.10).

Im Vereinbarungsteil des Hauptprogrammes können nur Ausdrücke für die Indexgrenzen auftreten, die zum Zeitpunkt der Vereinbarung auswertbar sind. Das Wortsymbol **packed** darf bei Vereinbarung eines dynamischen Feldtyps nicht auftreten, d. h. Sequenzen wie **packed dynamic array** oder **dynamic packed array** sind nicht zulässig.

2.3.2.5 Zeichenketten (Strings)

Ein spezieller, häufig vorkommender, als gepackt qualifizierter Array-Typ ist der statische String-Typ als Vektor mit dem Komponententyp char:

packed array [1..Laenge] **of** char;

mit der ganzzahligen Konstanten *Laenge* > 1.

Beispiel 2.3.17:

Die Vereinbarung

packed array [1..15] **of** char;

beschreibt Zeichenketten mit jeweils 15 Zeichen vom Typ *char*. Konstanten zu diesem Typ sind dann z. B.

'PASCALXSC SUPER' oder 'Stringkonstante',

wobei zwischen den beiden Apostrophen genau 15 Zeichen stehen.

2.3.2.6 Dynamische Strings

PASCAL–XSC

Für die vereinfachte Vereinbarung von Stringvariablen ist die Typangabe

string [Laenge]

oder nur

string

vorgesehen.

PASCAL–XSC

Dabei muß *Laenge* eine positive *integer*-Konstante sein, die implementierungsabhängig durch eine Maximallänge, z. B. 255, beschränkt ist. Fehlt die Längenangabe, so wird automatisch die Maximallänge eingesetzt. Der Wertebereich dieses dynamischen Stringtyps besteht aus allen Zeichenketten mit 0, 1, 2, .. *Laenge*−1, *Laenge* Zeichen.

Die tatsächliche Länge eines Wertes bezeichnet man als aktuelle Länge. Diese wird zur Laufzeit des Programms dynamisch verwaltet und kann mit der Funktion *length* abgefragt bzw. mit der Prozedur *setlength* verändert werden (vgl. Abschnitt 2.9).

Variablen vom Typ *string* können auch indiziert werden. So bezeichnet *s[i]* das *i*-te Zeichen der Zeichenkette *s* und ist vom Typ *char*. Zugriffe auf Komponenten außerhalb der vereinbarten Länge sind unzulässig. Die aktuelle Länge einer *string*-Variablen kann nur durch die Prozedur *setlength* oder durch eine Wertzuweisung an die *string*-Variable verändert werden.

Zur weiteren Verwendung von dynamischen Strings siehe Stringausdrücke (Abschnitt 2.4.3.2) und Textverarbeitung (Abschnitt 2.9).

2.3.2.7 Verbunde (Records)

Ein Record besteht aus einer festen Anzahl von Komponenten von jeweils beliebigem Typ. Er wird in der Form:

> **record** Datensatzliste **end**

vereinbart, wobei eine Datensatzliste aus einer Aufzählung von Komponenten der Form

> Namensliste: Typ; ...

besteht. Die Komponenten eines Records können als Variable (Komponentenvariable) verwendet werden, wobei der Zugriff folgende Form hat:

> Recordname.Komponentenname

Beispiel 2.3.18:

```
record
   stunde: 1..24;
   minute, sekunde : 1..60;
end

record
   re, im : real
end;
```

```
var datum: record monat: (jan, feb, mar, apr, mai, jun,
                          jul, aug, sep, okt, nov, dez);
                  tag:    1..31;
                  jahr:   integer;
           end;
```

Für die Variable *datum* sind die Komponentenvariablen zugreifbar durch

datum.monat   datum.tag   datum.jahr

2.3.2.8 Records mit Varianten

Ein Record kann um sogenannte Varianten erweitert werden, wobei die Abspeicherung der Varianten im gleichen Speicherbereich erfolgt. Die Kontrolle über diesen Bereich ist dem Programmierer überlassen. Damit ist die Übergabe von Werten ohne die in PASCAL vorgesehene strenge Typkontrolle möglich. Die Auflistung der Varianten bei der Typdefinition erfolgt im Anschluß an die festen Komponenten in der folgenden Form:

```
case
   Auswahlkomponente : { kann entfallen }
   Auswahltyp of
      Auswahlmarkenliste: (Datensatzliste); ... { nicht leer }
```

Dabei wird die Auswahlkomponente, die eigentlich noch zu den festen Komponenten zu zählen ist, durch einen Namen gekennzeichnet; der Auswahltyp legt den Typ der Auswahlkomponente und der nachfolgend aufgelisteten Auswahlmarken (Konstanten des Auswahltyps) fest. Als Auswahltyp sind die Typen *integer, boolean, char* und Aufzählungstypen sowie deren Unterbereiche zulässig.

Auf Komponenten einer Variante soll erst nach *Einschaltung* der gewünschten Variante zugegriffen werden. Darunter versteht man die Besetzung der Auswahlkomponente mit dem Wert der entsprechenden Auswahlmarke. Wurde bei der Vereinbarung auf die Angabe der Auswahlkomponente verzichtet, so wird eine Variante durch den ersten Zugriff auf eine ihrer Komponenten eingeschaltet.

Der Zugriff auf die Komponente einer Variante erfolgt wie auf feste Komponenten des Records.

Beispiel 2.3.19:

Ist ein Record-Typ *Schild* mit Variantenteil definiert durch

```
type
   Form = (Kreis, Rechteck, Dreieck);
   Schild = record
               Seriennr: integer;
               Material: (Blech, Kunststoff);
```

```
              Preis: real;
              case Figur: Form of
                 Kreis: (Radius : real);
                 Rechteck: (Seite1, Seite2: real);
                 Dreieck: (Grundseite, Winkellinks,
                               Winkelrechts: real);
           end;
```

so hat eine Variable s, vereinbart gemäß

```
var s : Schild;
```

drei Varianten, deren Komponenten man wie folgt ansprechen kann:

```
s.Figur   :=  Kreis;
s.Radius  :=  3.5;
```

oder etwa

```
s.Figur   :=  Rechteck;
s.Seite1  :=  4.8;
s.Seite2  :=  7.4;
```

oder

```
s.Figur         :=  Dreieck;
s.Grundseite    :=  5;
s.Winkellinks   :=  18.1;
s.Winkelrechts  :=  45;
```

PASCAL–XSC

Mit Ausnahme des Typs *string* darf kein dynamischer Feldtyp als Recordkomponente auftreten.

2.3.2.9 Mengen (Sets)

Der Wertebereich eines Mengentyps besteht aus allen Teilmengen eines vorgegebenen Grundbereichs. Die Typdefinition einer Menge muß deshalb nur den Typ des Grundbereichs enthalten:

set of Grundbereichstyp

Als Grundbereichstyp sind Unterbereiche von *integer*, *boolean*, *char* und Aufzählungstypen sowie die drei letzten Typen selbst zugelassen. Die meisten Implementierungen erlauben im Grundbereich nur Elemente x mit $0 \leq \text{ord}(x) \leq 255$. Ein Zugriff auf die Elemente einer Menge M analog den Komponentenvariablen bei Feldern und Records ist nicht vorgesehen. Dagegen kann für einen Wert x aus dem Grundbereich festgestellt werden, ob $x \in M$ gilt:

x **in** M

mit dem Ergebniswert *true* oder *false*.

Im einfachsten Fall kann man eine Menge dadurch erzeugen, daß man die gewünschten Elemente des Grundbereiches in folgender Form aufzählt:

[Elementeliste]

Die leere Menge gehört zu jedem Mengentyp und wird in der Form [ ] notiert.

Beispiel 2.3.20:

Der durch

type menge = **set of** 1..3;

vereinbarte Mengentyp enthält als Werte die Teilmengen

[ ], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]

Ein Zeichensatz

set of char

enthält als Werte z. B.

['a','b','c'] oder ['a'..'z', '0'..'9', '␣'].

Diese letzte Menge kann auch in der Form des Mengenausdrucks

['a'..'z'] + ['0'..'9'] + ['␣']

geschrieben werden.

2.3.2.10 Dateien (Files)

Ein File besteht aus einer Folge von beliebig vielen Komponenten gleichen Typs. In der Typdefinition genügt also die Angabe des Komponententyps:

file of Komponententyp

Der Komponententyp darf ein beliebiger Typ sein, ausgenommen ein File Typ bzw. ein dynamischer Feldtyp.

Ein direkter Zugriff auf die Komponenten eines Files analog den Komponentenvariablen bei Feldern und Records ist nicht vorgesehen. Hierzu steht eine sogenannte Puffervariable vom Komponententyp zur Verfügung, die automatisch mit der Vereinbarung einer Filevariablen f vereinbart ist. Die Puffervariable $f\uparrow$ erlaubt zu jedem Zeitpunkt den Zugriff auf eine einzelne Komponente des Files, die sogenannte aktuelle Komponente. Welche Komponente gerade aktuelle Komponente ist, hängt von den vorausgegangenen Fileoperationen wie *reset, rewrite, put* oder *get* ab:

rewrite(f) initialisiert f für nachfolgende Schreiboperationen. Die erste Komponente der Filevariablen f ist aktuelle Komponente. Es ist *eof(f) = true* und die Puffervariable $f\uparrow$ ist undefiniert.

put(f) schreibt den Wert von $f\uparrow$ in die aktuelle Komponente, die nachfolgende Komponente wird aktuelle Komponente, *eof(f) = true*. Die Puffervariable $f\uparrow$ ist undefiniert.

reset(f) initialisiert f für nachfolgende Leseoperationen. Die erste Komponente wird aktuelle Komponente. Falls *eof(f) = true* ist, ist das File leer und es kann nichts gelesen werden. In diesem Fall ist $f\uparrow$ undefiniert. Falls *eof(f) = false* ist, erhält die Puffervariable den Wert der aktuellen Komponente.

get(f) bewirkt, daß die der aktuellen Komponente nachfolgende Komponente zur neuen aktuellen Komponente und der Puffervariablen $f\uparrow$ zugewiesen wird, falls *eof(f) = false*. Andernfalls ist die Puffervariable $f\uparrow$ undefiniert.

Dabei ist *eof* (*end of file*) eine logische Funktion mit dem Wert *false*, wenn die aktuelle Komponente eine definierte Komponente des Files ist. Andernfalls liefert sie den Wert *true*.

Beschreiben oder Lesen eines Files sind also jeweils sequentielle Vorgänge, die stets bei der ersten Komponente eines Files beginnen. Während der Lesephase muß jeweils *eof(f) = false* gelten, in der Schreibphase *eof(f) = true*.

Beispiel 2.3.21:

```
var f : file of integer;
    ⋮
begin
  rewrite(f);             {Initialisieren für Beschreiben}
  for i:= 1 to 100 do     {das File f besteht aus}
  begin                   {100 Komponenten mit den}
    f↑ := i;              {Werten 1 bis 100}
    put(f)
  end;
   ⋮
  reset(f)                {Intialisieren für Lesen}
  while not eof(f) do     {Die Komponenten von f werden}
  begin
    writeln (sqr(f↑));    {der Reihe nach gelesen und die}
    get(f)                {Quadrate ihrer Werte ausgedruckt}
  end
end.
```

2.3.2.11 Textdateien (Textfiles)

Ein spezieller, häufig vorkommender File-Typ ist der Textfile-Typ *text*, der mit Komponententyp *char* vordefiniert ist. Textfiles können grundsätzlich in der gleichen Weise wie Files gehandhabt werden. Da Texte meist in einer Zeilenstruktur vorliegen, können Textfiles zusätzlich Zeilenendezeichen enthalten, die mit der logischen Funktion *eoln* (*end of line*) erkannt werden.

Falls für eine Textfilevariable t $eoln(t) = true$ gilt, so steht in der aktuellen Komponente das Zeilenendezeichen. Die Puffervariable $t\uparrow$ hat in diesem Fall den Wert $\sqcup$ (Leerzeichen).

Ein bequemes Arbeiten mit Textfilevariablen ist mit den Ein- und Ausgabeanweisungen *read, readln, write, writeln* möglich. In diesem Fall muß zusätzlich zur Ein- oder Ausgabeliste auch die Filevariable erwähnt werden (siehe Abschnitt 2.5.2). Falls er fehlt, wird automatisch die Standardtextfilevariable *input* bei der Eingabe oder *output* bei der Ausgabe angesprochen.

Beispiel 2.3.22:

```
var
   original, kopie: text;
   ch: char;
   ...
begin
   reset (original); rewrite (kopie);
   while not eof (original) do
   begin
      while not eoln (original) do
      begin
         read (original, ch);
         write (kopie, ch)
      end;
      readln (original);
      writeln (kopie)
   end;
end.
```

Es wird ein Text vom Original gemäß seiner Zeilenstruktur in eine Kopie übertragen.

2.3.3 Strukturierte arithmetische Standardtypen

PASCAL–XSC

In PASCAL–XSC stehen zusätzlich die arithmetischen Standardtypen *complex*, *interval*, *cinterval*, *rvector*, *cvector*, *ivector*, *civector*, *rmatrix*, *cmatrix*, *imatrix* und *cimatrix* zur Verfügung. Für diese gibt es keine Konstanten und innerhalb des Sprachkerns auch keine Operatoren und damit auch keine Ausdrücke. Diese sind erst nach Einbindung der entsprechenden Arithmetikmodule verwendbar (siehe Kapitel 3).

2.3.3.1 Der Typ *complex*

PASCAL–XSC

Für komplexe Zahlen der Form $z = x + iy$ mit dem Realteil x und dem Imaginärteil y (i ist die imaginäre Einheit), in Paarschreibweise $z = (x, y)$, kann in PASCAL der Typ *complex* eingeführt werden in der Form

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

Dieser Typ ist als Standardtyp in PASCAL–XSC vorhanden. Mit der Variablenvereinbarung

var z: complex;

kann man mittels *z.re* und *z.im* auf die Komponenten zugreifen.

2.3.3.2 Der Typ *interval*

PASCAL–XSC

Entsprechend dem Typ *complex* ist für reelle Intervalle

$$a = [\underline{a}, \overline{a}] := \{x \in I\!R \mid \underline{a} \leq x \leq \overline{a}\}$$

mit der unteren Schranke $\underline{a}$ und der oberen Schranke $\overline{a}$ der Standardtyp

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

definiert, wobei der Zugriff auf die Komponenten von

var a: interval;

mittels *a.inf* auf die untere und mittels *a.sup* auf die obere Schranke erfolgt.

2.3.3.3 Der Typ *cinterval*

PASCAL–XSC

Komplexe Intervalle sind als achsenparallele Rechtecke in der komplexen Ebene in der Form

type cinterval = **record** re, im : interval **end**;

definiert. Auch hier erfolgt der Zugriff auf die Komponenten von

var c: cinterval;

in der Form

c.re	für den Realteil (Projektion des Rechtecks auf die reelle Achse)
c.im	für den Imaginärteil (Projektion des Rechtecks auf die imaginäre Achse)

In diesem Fall sind die Komponentenvariablen ihrerseits Intervalle, auf deren reelle Schranken man wiederum mittels

c.re.inf   c.re.sup   c.im.inf   c.im.sup

zugreifen kann.

2.3.3.4 Die Vektor- und Matrixtypen

PASCAL–XSC

Für Vektoren und Matrizen mit Komponententyp *real*, *complex*, *interval* und *cinterval* stehen folgende dynamische Standardtypen zur Verfügung:

type	rvector	=	**dynamic array** [∗] **of** real;
	rmatrix	=	**dynamic array** [∗] **of** rvector;
	cvector	=	**dynamic array** [∗] **of** complex;
	cmatrix	=	**dynamic array** [∗] **of** cvector;
	ivector	=	**dynamic array** [∗] **of** interval;
	imatrix	=	**dynamic array** [∗] **of** ivector;
	civector	=	**dynamic array** [∗] **of** cinterval;
	cimatrix	=	**dynamic array** [∗] **of** civector;

2.3.4 Pointer (Zeiger)

Alle bisher besprochenen Datentypen von Standard-PASCAL sind statisch, d. h. alle Variablen zu diesen Typen werden zur Übersetzungszeit allokiert und ihre Anzahl bleibt während der Ausführung des Programms unverändert. Wünschenswert ist aber oft ein Mechanismus, der es gestattet, Variablen während der Ausführung eines Blockes zu erzeugen. Diese Lücke schließt die Pointervariable.

Eine Pointervariable p ist ein Zeiger (der Wert ist eine Adresse), der auf eine Variable $p\uparrow$ des zugehörigen referierten Typs verweist. Diese referierte Variable muß nicht vereinbart werden. Sie wird mittels der Standardprozedur *new* zur Laufzeit des Programms dynamisch erzeugt, während die Pointervariable selbst wie andere Variablen vereinbart wird.

Die Typdefinition muß also den referierten Typ enthalten:

type Pointertypname = ↑ Typname

Dabei darf der mit seinem Typnamen angegebene referierte Typ ein beliebiger Standard-PASCAL Typ sein. Im Unterschied zur bisherigen Festlegung, daß eine Größe vor ihrer Verwendung vereinbart sein muß, kann die Vereinbarung des referierten Typs auch erst nach der Vereinbarung des Pointers stehen. Die Werte eines Pointer-Typs sind also Verweise auf Variablen vom referierten Typ, erweitert um den Wert **nil** (Pointerkonstante), der zu jedem Pointer Typ gehört und auf keine referierte Variable zeigt. Außer dieser Konstanten **nil** sind keine weiteren Werte des Pointer-Typs explizit zugreifbar.

Beispiel 2.3.23:

```
type daten   = array [1..20] of real;
     zeiger  = ↑ element;
     element = record dat: daten;
                      nachfolger: zeiger
               end;
```

Die Erzeugung eines neuen referierten Elements geschieht durch Aufruf der Standardprozedur *new*:

new (PointerVariable);

Durch

```
type zeiger = ↑ reftyp;
     reftyp = record
                 ⋮
              end;
var p : zeiger;
  ⋮
new (p);
```

wird eine referierte Variable mit der Bezeichnung $p\uparrow$ vom Typ *reftyp* erzeugt und die Pointervariable p zeigt auf diese referierte Variable, ohne daß der Wert von p explizit bekannt wird.

Ist die referierte Variable von einem Record-Typ mit Varianten, so kann man durch Angabe einer Auswahlmarke beim Aufruf von *new* eine spezielle Variante erzeugen:

```
new (PointerVariable, Auswahlmarke);
```

Enthält diese Variante wiederum einen Variantenteil so kann man durch

```
new (PointerVariable, Auswahlmarke, ···, Auswahlmarke);
```

sukzessive alle geschachtelten Varianten spezifizieren.

Für Pointervariable sind Wertzuweisungen möglich

```
PointerVariable := Pointer Ausdruck;
```

wobei ein Pointerausdruck entweder die Konstante **nil**, eine Pointervariable oder ein Funktionsaufruf mit Ergebnis vom Typ Pointer darstellt. Damit ist zugleich ausgedrückt, daß man auch Funktionen mit Ergebnistyp Pointer formulieren kann. Es sind also auf zweiter Referenzstufe auch Funktionen mit strukturiertem Ergebnistyp möglich!

Für den Vergleich von Pointer-Ausdrücken sind nur die Vergleichsoperatoren = und <> erlaubt, z. B. $p =$ **nil** oder $p <> q$.

Mit Aufruf der Standardprozedur

```
dispose (PointerVariable);
```

wird der von der referierten Variablen $p\uparrow$ belegte Speicherplatz für weitere Verwendung freigegeben. Die Pointervariable p ist danach unbestimmt, ebenso wie alle anderen Verweise auf $p\uparrow$. Referierte Variable, die durch

```
new (p, m1, m2, ···, mk);
```

erzeugt wurden, müssen entsprechend durch

```
dispose (p, m1, m2, ···, mk);
```

freigegeben werden. Dabei müssen die Werte von *m1* bis *mk* beim Aufruf von *dispose* identisch mit den Werten beim Aufruf von *new* sein.

Beispiel 2.3.24:

```
var p, q: zeiger;
begin
    ⋮
  new (p);
  p↑.dat := {Wert ensprechend dem Typ daten};
  p↑.nachfolger := nil;
  arbeite (p); {Prozeduraufruf zur weiteren Verarbeitung}
    ⋮
  dispose(p); {Nicht mehr benötigten Speicherlatz wieder freigeben}
    ⋮
```

PASCAL–XSC

Als referierte Typen sind beliebige Typen zugelassen, ausgenommen dynamische Feldtypen.

Neben *dispose* ist eine weitere Möglichkeit zur Speicherplatzrückgabe mit den beiden Standardprozeduren *mark* und *release* verfügbar. Durch den Aufruf *mark (Pointervariable)* besteht die Möglichkeit, mit Ausführung des nachfolgenden Aufrufs der Standardprozedur *release (Pointervariable)* mit derselben, unveränderten Pointervariablen alle seit Ausführung von *mark* allokierten Speicherbereiche freizugeben. Danach hat die in *mark* und *release* verwendete Pointervariable einen unbestimmten Wert wie auch alle Referenzen auf den zurückgegebenen Speicherbereich.

Zu beachten ist, daß innerhalb eines Programms und aller darin verwendeten Module nur entweder mit *dispose* oder mit *mark/release* gearbeitet werden darf.

2.3.5 Verträglichkeit von Typen (Typkompatibilität)

Die Zulässigkeit von bestimmten Operationen ist im nachfolgenden durch die *Typverträglichkeit* (Kompatibilität) der betroffenen Operanden geregelt. Dabei heißen die Typen T1 und T2 *verträglich*, wenn gilt:

(a) T1 und T2 sind derselbe Typ.

(b) T1 ist ein Unterbereich von T2 oder T2 ist ein Unterbereich von T1 oder T1, T2 sind Unterbereiche desselben Grundtyps.

(c) T1, T2 sind Set-Typen über verträglichen Elementtypen und beide sind zugleich ungepackt oder gepackt.

(d) T1, T2 sind (statische) String-Typen mit derselben Komponentenzahl.

Speziell für die Wertzuweisung wird die Zulässigkeit über die *Zuweisungsverträglichkeit* des Typs T1 der linksstehenden Variablen mit dem Typ T2 des rechtsstehenden Ausdrucks festgelegt. Diese liegt vor, wenn gilt:

(a) T1 und T2 sind derselbe Typ, aber kein File-Typ.

(b) T1 ist Typ real, T2 ist Typ integer.

(c) T1 und T2 sind verträgliche endliche Typen (also *integer, boolean, char*, Aufzählungstypen oder Unterbereiche hiervon) und der Wert vom Typ T2 liegt in T1.

(d) T1, T2 sind verträgliche Set-Typen und die Elemente des Wertes vom Typ T2 liegen im Elementtyp von T1.

(e) T1, T2 sind verträgliche (statische) String-Typen.

Die Zuweisungsverträglichkeit wird auch beim Wertaufruf von formalen Argumenten einer Funktion oder Prozedur für den aktuellen Ausdruck angewendet. Beim Referenzaufruf wird dagegen die Typverträglichkeit beachtet.

Beispiel 2.3.25:

Mit der Vereinbarung

type
  vek1 = **array** [1..10] **of** real;
  vek2 = **array** [1..10] **of** real;
  vek3 = vek1;

sind *vek1* und *vek3* verträglich, da sie denselben Typ darstellen, während dies für *vek1* und *vek2* nicht gilt, obwohl sie dieselbe Struktur besitzen.

PASCAL–XSC

Wegen der Einführung dynamischer Typen und des dynamischen Stringkonzepts müssen auch die Verträglichkeitsfestlegungen erweitert werden. Diese Erweiterungen werden in den nachfolgenden Abschnitten erläutert.

Daneben gibt es die Möglichkeit durch Überladung des Zuweisungsoperators := (vgl. Abschnitt 2.7.12) die Zuweisungsverträglichkeit auf normalerweise nicht kompatible Typen auszudehnen. Diese „überladene Verträglichkeit“ gilt dann allerdings nur für die Zuweisung, nicht aber für einen Wertaufruf von Unterprogrammen.

2.3.5.1 Verträglichkeit von Array-Typen

PASCAL–XSC

Zunächst sind wie in Standard-PASCAL zwei Array-Typen nur dann *verträglich*, wenn es sich um dieselben Typen handelt. Dies heißt also, daß ein dynamischer Typ nicht mit einem statischen verträglich ist. Obwohl für die Wertzuweisung mehr Möglichkeiten denkbar wären, gilt weiter: Ein Wert des Array-Typs T2 ist *zuweisungsverträglich* mit der links stehenden Variablen vom Array-Typ T1, wenn

- beide Typen verträglich sind und die korrespondierenden Indexbereiche in ihrer Länge übereinstimmen oder

- T1 *anonym* ist und beide *strukturgleich* sind.

  Dabei spricht man von einem anonymen Typ einer Variablen, wenn dieser in Verbindung mit ihrer Vereinbarung kein expliziter Typname zugeordnet werden kann. Dies kann bei Komponentenvariablen (Teilfelder) der Fall sein (vgl. auch Abschnitt 2.3.2.2).

  Zwei Array-Typen heißen strukturgleich, wenn die Komponententypen gleich sind und die Indexbereiche in Anzahl, Länge und Basistyp übereinstimmen. Falls für einen Array Typ noch keine Indexbereiche spezifiziert sind, so erfüllt er jede Längenforderung. Diese Situation kann allerdings nur für den Typ eines formalen Arguments zutreffen.

Die Wertzuweisung (vgl. Abschnitt 2.5.1) ist also in folgenden Fällen ohne weiteres möglich:

Typ der linken Seite	Typ der rechten Seite	Wertzuweisung zulässig
dynamisch anonym	bel. Array Typ	falls strukturgleich
dynamisch bekannt	dynamisch bekannt	falls gleiche Typen
statisch anonym	bel. Array Typ	falls strukturgleich
statisch bekannt	statisch bekannt	falls gleiche Typen

PASCAL–XSC

In den verbleibenden Fällen ist eine Wertzuweisung nur unter gleichzeitiger *Qualifikation* der rechten Seite durch

Arraytypname (Arrayausdruck)

möglich. Dabei dient der Arraytypname als Typanpassungsfunktion (vgl. Abschnitt 2.4.3.1). Allerdings kann ein Arrayausdruck mit Typ T2 mittels eines Namens für einen Arraytyp bzw. einen dynamischen Arraytyp T1 nur dann qualifiziert werden, wenn T1 und T2 strukturgleich sind.

Beispiel 2.3.26:

Die durch

```
const grad = ...;
type  poly = dynamic array [*] of real;
      vec  = dynamic array [*] of real;
```

vereinbarten Typen *poly* und *vec* sind nicht verträglich. Falls eine Vektoraddition für den Typ *vec* als Operator vorhanden ist, so können die durch

```
var p, q: poly[0..grad];
```

vereinbarten Polynome p und q mit Hilfe der Qualifikation addiert werden durch

```
p := poly (vec (p) + vec (q));
```

2.3.5.2 Verträglichkeit von Zeichenketten

In Standard-PASCAL sind Zeichenkettentypen (in folgendem *array*-Zeichenkettentypen genannt) verträglich und zuweisungsverträglich, falls nur ihre Längen übereinstimmen.

PASCAL–XSC

Diese Regelung wird auch für die mit dem Standardnamen *string* eingeführten neuen Zeichenkettentypen (im folgenden *string*-Zeichenkettentypen genannt) beibehalten. Für die *string*-Zeichenkettentypen gilt:

- Zwei *string*-Zeichenkettentypen sind grundsätzlich *verträglich*. Ein *string*-Zeichenkettentyp ist mit keinem weiteren Typ *verträglich*.

- Ein Zeichenkettenwert vom Typ T2 ist *zuweisungsverträglich* zu einer Variablen von Typ T1, falls T1 ein *string*-Zeichenkettentyp und T2 ein *array*-Zeichenkettentyp, *string*-Zeichenkettentyp oder ein *char*-Typ ist.

2.4 Ausdrücke

In diesem Abschnitt soll das Ausdruckskonzept von PASCAL beschrieben werden. Für die zusätzlichen Datentypen von PASCAL–XSC werden Ergänzungen angegeben. Die Möglichkeiten von PASCAL–XSC, eigene Operatoren und Funktionen mit beliebigem Ergebnistyp zu definieren, erlauben darüberhinaus, für beliebige benutzerdefinierte Typen ein eigenes Ausdruckskonzept zu schaffen, das gemäß den üblichen Regeln von Prioritätsstufen und Klammerstruktur verarbeitet wird.

Nachfolgend werden zur abkürzenden Kennzeichnung von Typen die folgenden Kurzschreibweisen verwendet

A	Array, Feld
B	*boolean*
CH	*char*
CD	Code Typ, Aufzählungstyp
CIR	*cinterval* (komplexe Intervalle)
CR	*complex* (komplexe Zahlen)
DOT	*dotprecision*
F	File, Datei
I	*integer*
IR	*interval* (reelle Intervalle)
MCIR	*cimatrix* (komplexe Intervallmatrizen)
MCR	*cmatrix* (komplexe Matrizen)
MIR	*imatrix* (reelle Intervallmatrizen)
MR	*rmatrix* (reelle Matrizen)
P	Pointer
R	*real*
REC	Record
ST	*string*, Zeichenkette
TF	Textfile, *text*
VCIR	*civector* (komplexe Intervallvektoren)
VCR	*cvector* (komplexe Vektoren)
VIR	*ivector* (reelle Intervallvektoren)
VR	*rvector* (reelle Vektoren)

Daneben werden häufig die von den Syntaxdiagrammen her stammenden Abkürzungen

AUSD	für Ausdruck
KONST	für Konstante
OPD	für Operand
VAR	für Variable

verwendet.

2.4.1 Standardausdrücke

Ausdrücke zu den Standardtypen *integer*, *real* und *boolean* werden wie üblich formuliert, d. h. Operanden entsprechenden Typs werden durch Operatoren verküpft. Die Auswertung eines Ausdrucks geschieht entsprechend der Priorität der Operatoren (höhere vor niederer Priorität), bei Operatoren gleicher Priorität von links nach rechts. Eine vorhandene Klammerstruktur geht in jedem Fall vor. Der Ergebnistyp des Ausdrucks wird durch den entsprechend den Prioritätsregeln zuletzt auszuführenden Operator festgelegt. Ein Ausdruck hat grundsätzlich den Aufbau:

mon. Operator
Operand {nicht leer}
dyad. Operator Operand ...

Ein Operand kann dabei auch durch einen geklammerten Ausdruck oder einen Funktionsaufruf gegeben sein. Dabei ist zu beachten, daß nicht für jede mögliche Kombination von Operanden auch Operatoren verfügbar sind und daß deren Bedeutung von den Operanden abhängt:

Monadischer Operator	Operandentyp	Ergebnistyp
+, −	I, R	I, R
not	B	B

Dyadischer Operator	Operandentyp	Ergebnistyp
+, −, ∗, /, **div**, **mod**	I	I (R bei /)
+, −, ∗, /	I und R	R
+, −, ∗, /	R	R
or, **and**	B	B
+, −, ∗	SET	SET
=, <>, <, >, <=, >=	I, R, CH, CD, ST	B
=, <>, <=, >=	SET, B	B
in	linker Operand: I, B, CH und CD rechter Operand: entspr. SET Typ	B

Für Operanden vom Mengentyp (SET) bedeutet + die Mengenvereinigung, − die Mengendifferenz, ∗ den Mengendurchschnitt. Der Vergleichsoperator **in** bedeutet Mitgliedschaft (Element von), <= bzw. >= bedeuten den Vergleich auf Teilmengen- bzw. Obermengeneigenschaft.

Als Operanden können auftreten:

Konstanten, z. B.	3.14159
Variablen, z. B.	x, t
Funktionsaufrufe, z. B.	funk(x,t)
Standardfunktionsaufrufe, z. B.	sin(x), sqrt(y)
geklammerte Ausdrücke, z. B.	(x+y)

In Standard-PASCAL sind die Operatoren in folgende *Prioritätsstufen* eingeteilt:

Priorität	dyadische Operatoren	monadische Operatoren
0 (niederste)	$=,\ <>,\ <=,\ >=,\ <,\ >$, **in**	
1	$+,\ -$, **or**	$+,\ -$
2	**mod**, **div**, $*$, $/$, **and**	
3 (höchste)		**not**

PASCAL–XSC

Im Unterschied zu Standard-PASCAL haben die monadischen Operatoren $+$ und $-$ die höchste Priorität 3 und es können mehrere monadische Operatoren unmittelbar aufeinanderfolgen:

```
        mon. Operator ...
                            Operand   {nicht leer}
                                      dyad. Operator   Operand   ...
```

Unmittelbar aufeinanderfolgende monadische Operatoren werden von rechts nach links ausgeführt. Zusätzlich zu den Standard-PASCAL-Operatoren stehen folgende Operatoren bereit:

Dyadischer Operator	Operandentyp	Ergebnistyp
$+<,\ -<,\ *<,\ /<$	I oder R	R
$+>,\ ->,\ *>,\ />$	I oder R	R
$+$	CH oder ST	ST
in	linker Operand: ST oder CH rechter Operand: ST	B

Darüber hinaus stehen die dyadischen Operatorzeichen $**$, $+*$ und $><$ zur Verfügung, die als Operatoren erst bei Verwendung der Arithmetikmodule (vgl. Kapitel 3) eine vordefinierte Bedeutung erhalten.

Die Operatoren sind in PASCAL–XSC in folgende *Prioritätsstufen* eingeteilt:

Priorität	dyadische Operatoren	monadische Operatoren
0 (niederste)	$=,\ <>,\ <=,\ >=,\ <,\ >$, **in**, $><$	
1	**or**, $+,\ +<,\ +>,\ -,\ -<,\ ->,\ +*$	
2	$*,\ *<,\ *>,\ /,\ /<,\ />,\ **$ **mod**, **div**, **and**,	
3 (höchste)		$+,\ -$, **not**

2.4.1.1 Ganzzahliger Ausdruck

Er besteht aus ganzzahligen Operanden, die mittels der ganzzahligen Operatoren $+, -, *$, **div** (ganzzahliger Anteil bei der Division) und **mod** (Rest bei der ganzzahligen Division) verknüpft werden. Es sind folgende Standardfunktionen verfügbar:

Standardfunktion	Bedeutung
trunc (R AUSD)	Rundung durch Abschneiden der Nachkommastellen
round (R AUSD)	Rundung zur nächsten ganzen Zahl, es ist $round\ (r) = \begin{cases} trunc\ (r + 0.5) & \text{für } r \geq 0 \\ trunc\ (r - 0.5) & \text{für } r < 0 \end{cases}$
ord (AUSD) B, CD, CH	Ordnungszahl des angegebenen Elementes. Die Elemente dieser Typen sind beim kleinsten Element mit der 0 beginnend in ihrer natürlichen Ordnung durchnumeriert.
ord (I AUSD)	Identität, d. h. $ord(v) = v$
succ (I AUSD)	Nachfolger, $succ(v) = v + 1$
pred (I AUSD)	Vorgänger, $pred(v) = v - 1$
abs (I AUSD)	Absolutbetrag
sqr (I AUSD)	Quadrat (square), $sqr(v) = v^2$

PASCAL–XSC

Weitere *integer*-Standardfunktionen sind:

Standardfunktion	Bedeutung
loc (VAR)	Implementierungsabhängige Adresse der Variablen
ord (P AUSD)	Implementierungsabhängiger Wert des Arguments
sign (AUSD) I, R, DOT	Vorzeichen, $sign\ (a) = \begin{cases} -1 & \text{für } a < 0 \\ 0 & \text{für } a = 0 \\ +1 & \text{für } a > 0 \end{cases}$
lbound (A VAR, I KONST)	Untere Grenze eines Indexbereichs
ubound (A VAR, I KONST)	Obere Grenze eines Indexbereichs
expo (R AUSD)	Exponentenanteil bezogen auf normalisierte Mantisse (vgl. Abschnitt 2.4.1.2)

Für eine Array-Variable A liefert *lbound(A,n)* die untere Grenze des Indexbereichs der n-ten Dimension. Bei fehlendem n (I Konstante) wird die erste Dimension angesprochen. Entsprechendes gilt für *ubound*. Als abkürzende Schreibweisen sind *lb* (für *lbound*) und *ub* (für *ubound*) zugelassen.

Darüber hinaus existiert die Funktion *ival* zur Wandlung von Strings in *integer*-Werte (ausführlich beschrieben im Abschnitt 2.9).

2.4.1.2 Reeller Ausdruck

Er besteht aus reellen oder ganzzahligen Operanden, die mittels der Gleitkommaoperationen $+, -, *, /$ verknüpft werden. Bei der Verknüpfung zweier *integer*-Operanden mittels $+, -, *$ wird die ganzzahlige Operation ausgeführt. Es sind folgende Standardfunktionen verfügbar:

Standardfunktion	Bedeutung
abs (R Ausdruck)	Absolutbetrag $\|x\|$
sqr (R Ausdruck)	Quadrat x^2
sin (R Ausdruck)	Sinus $\sin x$
cos (R Ausdruck)	Kosinus $\cos x$
arctan (R Ausdruck)	Arkustangens $\arctan x$
exp (R Ausdruck)	Exponentialfunktion e^x
ln (R Ausdruck)	natürlicher Logarithmus $\ln x,\ x > 0$
sqrt (R Ausdruck)	Quadratwurzel $\sqrt{x},\ x \geq 0$

Beispiel 2.4.1:

Mit den Vereinbarungen

var x, y, v, w : real;
i, j : integer;

sind

sqr(x) + sin(y+1.5)/ln(sqr(v)+sqr(w)+1.2) und
i **div** j + 1e−10

real-Ausdrücke.

PASCAL–XSC

Es stehen Gleitkommaoperationen für drei verschiedene Rundungen zur Verfügung. Die folgenden Ausführungen geben einen kurzen Überblick über die Grundlagen für die Benutzung dieser Operationen.

Ein Gleitkommasystem R ist charakterisiert durch eine Basis b (etwa 2 oder 10), eine endliche Anzahl n von Mantissenstellen (etwa 13) und einen Exponentenbereich mit dem kleinsten (*emin*) und dem größten (*emax*) zulässigen Exponenten. Eine normalisierte Gleitkommazahl x läßt sich darstellen als

$$x = \pm 0.d_1 d_2 ... d_n \cdot b^{\mathrm{ex}}$$

wobei $d_1 \neq 0$, $0 \leq d_i \leq b - 1$ und $emin \leq ex \leq emax$. Ein Gleitkommasystem läßt sich also charakterisieren als $R = R(b, n, emin, emax)$.

Ein Gleitkommasystem (vgl. auch [24] und [20]) ist bezüglich der arithmetischen Operationen $+, -, *, /$ nicht abgeschlossen, d. h. die Verknüpfung zweier Gleitkommazahlen liefert eine reelle Zahl, die im allgmeinen keine Gleitkommazahl ist, z. B. mit $x = 0.58, y = 0.47$ aus $R(10, 2, -10, 10)$ ergibt $x + y = 1.05$. Diese Zahl liegt nicht im vorgegebenen Raster. Wenn man damit weiterrechnen will, muß man diese Zahl in das vorhandene Raster abbilden.

PASCAL–XSC

Das Beste, was man dabei tun kann, ist, auf eine der benachbarten Rasterzahlen abzubilden (zu runden). Eine solche Operation heißt auf ein Ulp (Unit in the last place, Einheit in der letzten Stelle) genau. Bildet man auf die nächstkleinere Gleitkommazahl ab, so bezeichnen wir das als optimale Rundung nach unten, bildet man auf die nächstgrößere Gleitkommazahl ab, so heißt dies optimale Rundung nach oben.

Den kleinsten lokalen Fehler begeht man, wenn man zur nächstgelegenen Gleitkommazahl rundet (1/2 Ulp). In PASCAL–XSC wird diese implementierungsabhängige Rundung zur nächstgelegenen Gleitkommazahl mit den üblichen Zeichen $+, -, *, /$ angesprochen, die optimale nach unten bzw. oben gerichtete Rundung wird mit den Zeichen +<, −<, *<, /< bzw. +>, −>, *>, /> gekennzeichnet.

Die gerichteten Operationen sind notwendig, wenn man garantierte Schranken für den Wert eines *real*-Ausdrucks bestimmen will. Man muß sich dann allerdings bei jeder Operation überlegen, in welche Richtung gerundet werden muß, damit der Gesamtausdruck in der gewünschten Weise abgeschätzt wird. Dabei ist bei auftretenden Literalkonstanten auch auf deren Rundung (Konvertierung) zu achten.

Die gerichteten Operationen werden auch für die Implementierung einer Intervallarithmetik benötigt, bei der das reelle Ergebnisintervall nach außen zum einschließenden Maschinenintervall gerundet werden muß, d. h. die untere Grenze ist nach unten, die obere Grenze nach oben zu runden.

Beispiel 2.4.2:

Die *real*-Ausdrücke 1/3 bzw. 1/<3 bzw. 1/>3 liefern in $R(10, 4, -5, 5)$ die Werte 0.3333 bzw. 0.3333 bzw. 0.3334.

Soll von dem Ausdruck $x \cdot y - v \cdot w$ eine untere bzw. eine obere Schranke berechnet werden, so hat der entsprechende Ausdruck in PASCAL–XSC die Form

x *< y −< v *> w   bzw.   x *> y −> v *< w .

Dabei gilt es zu beachten daß bei der Multiplikation im Ausdruck für den zweiten Operanden der Subtraktion jeweils in die entgegengesetzte Richtung zu runden ist.

PASCAL–XSC stellt die Standardfunktionen *succ* und *pred* nicht nur für *integer*- sondern auch für *real*-Argumente zur Verfügung. Die Aufrufe

succ (R Ausdruck)   und   pred (R Ausdruck)

liefern jeweils die nächstgrößere und nächstkleinere Gleitkommazahl.

PASCAL–XSC

Als weitere Standardfunktionen wird ein Satz von mathematischen Standardfunktionen zur Verfügung gestellt:

Standardfunktion	Bedeutung
exp2 (R Ausdruck)	Exponentiation zur Basis 2 2^x
exp10 (R Ausdruck)	Exponentiation zur Basis 10 10^x
log2 (R Ausdruck)	Logarithmus zur Basis 2 $\log_2 x$
log10 (R Ausdruck)	Logarithmus zur Basis 10 $\log_{10} x$
tan (R Ausdruck)	Tangens $\tan x$
cot (R Ausdruck)	Kotangens $\cot x$
arcsin (R Ausdruck)	Arkussinus $\arcsin x$
arccos (R Ausdruck)	Arkuskosinus $\arccos x$
arccot (R Ausdruck)	Arkuskotangens $\operatorname{arccot} x$
arctan2 (R Ausdruck, R Ausdruck)	arctan2(r1,r2) = arctan (r1/r2)
sinh (R Ausdruck)	Hyperbolischer Sinus $\sinh x$
cosh (R Ausdruck)	Hyperbolischer Kosinus $\cosh x$
tanh (R Ausdruck)	Hyperbolischer Tangens $\tanh x$
coth (R Ausdruck)	Hyperbolischer Kotangens $\coth x$
arsinh (R Ausdruck)	Areasinus $\operatorname{arsinh} x$
arcosh (R Ausdruck)	Areakosinus $\operatorname{arcosh} x$
artanh (R Ausdruck)	Areatangens $\operatorname{artanh} x$
arcoth (R Ausdruck)	Areakotangens $\operatorname{arcoth} x$

Alle in PASCAL–XSC verfügbaren reellen mathematischen Standardfunktionen liefern ein Ergebnis von maximaler Genauigkeit in dem Sinne, daß zwischen dem exakten reellen Ergebnis und der berechneten Gleitkommazahl keine weitere Gleitkommazahl liegt (1 Ulp Genauigkeit).

Darüber hinaus existiert die Funktion *rval* zur Wandlung von Strings in *real*-Werte (ausführlich beschrieben im Abschnitt 2.9) und ein Satz von Zerlegungs- und Kompositionsfunktionen für den Datentyp *real* (vgl. auch *expo* in Abschnitt 2.4.1.1):

Standardfunktion	Bedeutung
mant (R Ausdruck)	normalisierte Mantisse m von r, wobei der Wertebereich von m implementierungsabhängig ist
comp (R Ausdruck, I Ausdruck)	Komposition einer Mantisse (Typ R) und eines Exponenten (Typ I) zu einer *real*-Zahl, wobei die Mantisse nur in einem implementierungsabhängigen Wertebereich liegen darf

PASCAL–XSC

Beispiel 2.4.3:

Der Zusammenhang der Funktionen *mant*, *expo* und *comp* ist über die Beziehung

x = comp (mant (x), expo (x))

erklärt. Je nach Implementierung ergibt sich zum Beispiel:

Anweisung			Ergebnis		
m	:=	mant (100)	m	=	0.1
e	:=	expo (100)	e	=	3
x	:=	comp (m,e)	x	=	100 = 0.1E+03

2.4.1.3 Logischer Ausdruck

Er enthält neben den logischen Konstanten (Literalkonstante *true* und *false*) als Operanden Variable, logische Funktionen, Vergleiche, geklammerte logische Ausdrücke und die folgenden Standardfunktionen:

Standardfunktion	Bedeutung
pred (B Ausdruck)	Vorgänger bezüglich der Ordnung *false* < *true*
succ (B Ausdruck)	Nachfolger bezüglich der Ordnung *false* < *true*
odd (I Ausdruck)	der ganzzahlige Wert des Arguments ist ungerade (*true*) oder gerade (*false*)
eof (File Variable)	File-Ende erreicht (*true*) oder nicht (*false*)
eoln (Textfile Variable)	Zeilenende erreicht (*true*) oder nicht (*false*)

Dabei ist zu beachten, daß Vergleiche als Operanden zu klammern sind. Die Vergleichsoperatoren haben die übliche Bedeutung, das Zeichen <> bedeutet $\neq$ (ungleich). Bei Vergleichen von logischen Operanden bedeutet <= bzw. >= die logische Implikation $\rightarrow$ bzw. $\leftarrow$ und das Zeichen = bedeutet die logische Äquivalenz.

PASCAL–XSC

Weitere Standardfunktionen:

Standardfunktion	Bedeutung
lbound (Array Variable, I Konstante)	Untere Grenze eines Indexbereichs
ubound (Array Variable, I Konstante)	Obere Grenze eines Indexbereichs

Für eine Array-Variable A mit Indextyp *boolean* liefert *lbound(A,n)* die untere Grenze des Indexbereichs der n-ten Dimension. Bei fehlendem n (I Konstante) wird die erste Dimension angesprochen. Entsprechendes gilt für *ubound*. Als abkürzende Schreibweisen sind *lb* (für *lbound*) und *ub* (für *ubound*) zugelassen.

PASCAL–XSC

Vergleiche von Werten der arithmetischen Datentypen

complex, interval, cinterval

sowie

rvector,	*cvector,*	*ivector,*	*civector,*
rmatrix,	*cmatrix,*	*imatrix,*	*cimatrix*

sind, unter Verwendung der entsprechenden Arithmetikmodule, ebenfalls möglich. Eine genaue Darstellung findet man in Kapitel 3 (Arithmetikmodule).

Vergleiche von Werten vom Typ *dotprecision* sind direkt nicht möglich. Sie müssen mit Hilfe der *sign*-Funktion vorgenommen werden:

$$\text{sign (d)} := \begin{cases} 1 & \text{für} \quad \text{d} > 0 \\ 0 & \text{für} \quad \text{d} = 0 \\ -1 & \text{für} \quad \text{d} < 0 \end{cases}$$

Dabei ist d ein Ausdruck vom Typ *dotprecision*.

2.4.1.4 Zeichen-Ausdruck

Er enthält keine Operatoren, sondern nur eine Konstante, eine Variable oder einen Funktionsaufruf. Als Standardfunktionen existieren:

Standardfunktion	Bedeutung
pred (CH Ausdruck)	Vorgänger bezüglich der Ordnung im CH Typ
succ (CH Ausdruck)	Nachfolger bezüglich der Ordnung im CH Typ
chr (I Ausdruck)	Zeichen mit der Ordnungszahl des I Ausdrucks

PASCAL–XSC

Weitere Standardfunktionen:

Standardfunktion	Bedeutung
lbound (Array Variable, I Konstante)	Untere Grenze eines Indexbereichs
ubound (Array Variable, I Konstante)	Obere Grenze eines Indexbereichs

Für eine Array-Variable A mit Indextyp *char* liefert *lbound(A,n)* die untere Grenze des Indexbereichs der n-ten Dimension. Bei fehlendem n (I Konstante) wird die erste Dimension angesprochen. Entsprechendes gilt für *ubound*. Als abkürzende Schreibweisen sind *lb* (für *lbound*) und *ub* (für *ubound*) zugelassen.

2.4.1.5 Code-Ausdruck

Er enthält neben den Code-Konstanten und Variablen noch Funktionsaufrufe und die Standardfunktionen:

Standardfunktion	Bedeutung
pred (CD Ausdruck)	Vorgänger bezüglich der Ordnung im CD Typ
succ (CD Ausdruck)	Nachfolger bezüglich der Ordnung im CD Typ

PASCAL–XSC

Weitere Standardfunktionen:

Standardfunktion	Bedeutung
lbound (Array Variable, I Konstante)	Untere Grenze eines Indexbereichs
ubound (Array Variable, I Konstante)	Obere Grenze eines Indexbereichs

Für eine Array-Variable A mit einem Aufzählungstyp (Codetyp) als Indextyp liefert *lbound(A,n)* die untere Grenze des Indexbereichs der n-ten Dimension. Bei fehlendem n (I Konstante) wird die erste Dimension angesprochen. Entsprechendes gilt für *ubound*. Als abkürzende Schreibweisen sind *lb* (für *lbound*) und *ub* (für *ubound*) zugelassen.

Beispiel 2.4.4:

```
type niederschlag = (regen, hagel, schnee);
var  n: niederschlag;
     nmenge: array [niederschlag] of real;
       ⋮
  n:= succ (regen); { n erhält den Wert hagel }
  n:= ubound (nmenge); { n erhält den Wert schnee }
```

2.4.2 Ausdrücke mit genauer Auswertung (#-Ausdrücke)

PASCAL–XSC

Die in fast allen Programmiersprachen üblichen *real*-Ausdrücke haben den Vorteil einer einfachen Auswertung, indem je zwei Operanden verknüpft und gleich wieder ins vorgegebene *real*-Format gerundet werden. Diese Rundung bei jeder Einzeloperation kann jedoch das Endergebnis total verfälschen. Zur Vermeidung solcher unkontrollierbaren Effekte wurde in PASCAL–XSC das Konzept des *dotprecision*-Ausdrucks oder auch Lattenkreuzausdrucks eingeführt.

PASCAL–XSC

Syntaktisch werden Lattenkreuzausdrücke (#-Ausdrücke) stets mit einem # (Lattenkreuz) eingeleitet und haben eine der drei folgenden Formen:

    # (R GEN AUSD)   { exakte Übergabe (*dotprecision*) }

oder

    #* (R GEN AUSD)   { zur nächsten,   }
    #< (R GEN AUSD)   { zur nächstkleineren,   }
    #> (R GEN AUSD)   { zur nächstgrößeren   }
                      { *real*-Zahl gerundet   }

oder

    ## (R GEN AUSD)   { zum kleinsten umfassenden Intervall   }
                      { gerundet, Ergebnistyp *interval*   }

Innerhalb eines solchen Ausdrucks steht ein sogenannter GENAU AUSZUWERTENDER AUSDRUCK (GEN AUSD), der stets exakt ausgewertet wird. Dieser hat den syntaktischen Aufbau

    ± Summand
              Operator Summand ...

wobei nur die Operatoren + und − für die exakte (fehlerfreie) Addition und Subtraktion der folgenden Summanden zulässig sind:

DOT VAR	*dotprecision*-Variable
R OPD	*real*-Operand mit den Möglichkeiten I VAR I KONST R VAR R KONST
R OPD $*$ R OPD	exaktes, doppelt langes Produkt zweier R Operanden
(R GEN AUSD)	geklammerter genau auszuwertender Ausdruck
for i:= a **to** e **sum** (R GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a, e* I Ausdrücke sind
for i:= a **downto** e **sum** (R GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a, e* I Ausdrücke sind

PASCAL–XSC

Achtung: Die Operatoren $+, -, *$ haben innerhalb des GEN AUSD die Bedeutung der mathematisch exakten (ungerundeten) Operationen. Sie können deshalb *nicht* durch selbstdefinierte Operatoren überladen werden.

Die Laufanweisung mit **sum** dient zur abkürzenden Schreibweise von Summen. Dabei kann der GEN AUSD in der Klammer auch von der Laufvariable i abhängen. Ein Ausdruck der Form

$$\text{GEN AUSD}_a + \text{GEN AUSD}_{a+1} + ... + \text{GEN AUSD}_{e-1} + \text{GEN AUSD}_e$$

kann dadurch abgekürzt als

for i:= a **to** e **sum** (GEN AUSD$_i$)

geschrieben werden (vgl. auch Abschnitt 2.5.8.3 Laufanweisung). Eine leere Laufanweisung entspricht einem Summanden mit Wert Null. Entsprechendes gilt bei der Verwendung von **downto**.

In den *integer*-Ausdrücken a und e dürfen dabei nicht selbst wieder explizit #-Ausdrücke auftreten.

Beispiel 2.4.5:

Die Berechnung eines Skalarprodukts der Form

$$s := \sum_{i=1}^{10} a_i * b_i$$

kann in PASCAL–XSC mit nur einer Rundung erfolgen. Mit den Vereinbarungen

```
var  a, b  :  array [1..10] of real;
     s     :  real;
     d     :  dotprecision;
     i     :  integer;
```

kann man dies realisieren durch

```
d := # (0);
for i:=1 to 10 do
  d := # (d + a[i]*b[i]);
s := #* (d); { Rundung zur nächstgelegenen real-Zahl }
```

Die alternative abkürzende Schreibweise für dieses Programmstück wäre

```
s := #* (for i:=1 to 10 sum (a[i]*b[i]));
```

PASCAL–XSC

Beispiel 2.4.6:

Will man für den Wert des Ausdrucks $x * y - v * w$ die nächstgelegene bzw. nächstkleinere bzw. nächstgrößere Gleitkommazahl berechnen, so kann dies durch

#* (x * y − v * *w*) bzw.
#< (x * y − v * *w*) bzw.
#> (x * y − v * *w*)

erfolgen.

Achtung: Bei Verwendung von Literalkonstanten als *real*-Operanden innerhalb eines #-Ausdrucks ist zu beachten, daß diese zunächst in das interne *real*-Format konvertiert werden und so, je nach Implementierung, unvermeidbare Eingangsfehler entstehen können. So liefert z. B. der Ausdruck

(0.1)

bei interner Dualdarstellung nicht etwa eine kleinste Einschließung der reellen Zahl 0.1, sondern ein Punktintervall, das dem Wert der konvertierten Konstanten entspricht. Eine Einschließung für den reellen Wert 0.1 kann aber durch

intval ((<0.1) , (>0.1))

erzeugt werden (siehe auch Abschnitt 3.2).

2.4.3 Ausdrücke für die strukturierten Datentypen und Pointer-Ausdruck

Von den strukturierten Datentypen bieten in Standard-PASCAL nur Mengen die Möglichkeit, Ausdrücke der gewohnten Form mit Operatoren zu bilden. Ausdrücke für Felder und Records enthalten keine Operatoren, und für Files und Textfiles gibt es keine Ausdrücke.

PASCAL–XSC

Das Operatorkonzept in PASCAL–XSC (vgl. Abschnitt 2.7.6) gestattet es, Operatoren für beliebige Standarddatentypen und selbstdefinierte Datentypen zu vereinbaren. Damit hat der Benutzer die Möglichkeit, ein komplettes Ausdruckskonzept für diese Datentypen zu implementieren.

PASCAL–XSC

Für die strukturierten arithmetischen Standardtypen *complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix* und *cimatrix* gibt es bereits wie für *real*-Ausdrücke ein solches Ausdruckskonzept mit vielen vordefinierten Standardoperatoren und Standardfunktionen. Sie sind im Kapitel 3 (Arithmetikmodule) im entsprechenden Modul C_ARI, I_ARI, CI_ARI, MV_ARI, MVC_ARI, MVI_ARI und MVCI_ARI beschrieben.

2.4.3.1 Array-Ausdruck

Ein Array-Ausdruck enthält keine Operatoren. Er besteht damit nur aus Variablen.

PASCAL–XSC

Der Array-Ausdruck in PASCAL–XSC kann selbstdefinierte Operatoren und als Operanden neben Variablen und Funktionsaufrufen mit Array Ergebnistyp auch die sogenannte *Qualifikation* enthalten. Diese hat die Form

Array-Typname (Array-Ausdruck)

wobei der Array-Typname als Typanpassungsfunktion aufgefaßt werden kann, d. h. bei Strukturgleichheit erhält der Array-Ausdruck den mit dem Array-Typnamen bezeichneten Typ. Es gibt keine vordefinierten Standardoperatoren für die Verknüpfung dieser Operanden.

Beispiel 2.4.7:

```
type
   vektor = array [1..n] of real;
   polynom = array [0..n−1] of real;
var
   v : vektor;
   p : polynom;
...
   p := polynom (v);
   v := vektor (p);
```

Sinnvolle Anwendungen der Typanpassungsfunktion ergeben sich hauptsächlich im Zusammenhang mit dynamischen Feldern und Operatorkonzept (Abschnitt 2.7.6).

Der dynamische Array-Ausdruck hat denselben Aufbau wie der Array-Ausdruck, wobei anstelle von Arrays eben dynamische Arrays auftreten.

2.4.3.2 String-Ausdruck

Für Strings gibt es in Standard-PASCAL keine Operatoren und keine Standardfunktionen. Ein String-Ausdruck ist deshalb entweder eine String-Konstante oder eine String-Variable.

PASCAL–XSC

Auf der Grundlage des dynamischen String-Typs mit dem Standardnamen *string* können String-Operanden mit dem Standardoperator + zusammengesetzt werden:

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

Dabei wird der String *b* hinter den String *a* angefügt. Die aktuelle Länge des Ergebnisses ergibt sich aus der Summe der aktuellen Längen von *a* und *b*. Die Wirkung bei der Überschreitung der Maximallänge des Typs *string* ist implementierungsabhängig.

Als Operanden sind neben *string*-Konstanten und *string*-Variablen auch *string*-Funktions- und Standardfunktionsaufrufe (siehe Abschnitt 2.9 Textverarbeitung) und geklammerte *string*-Ausdrücke zulässig. Als Sonderfall eines *string*-Operanden kann auch ein Zeichen-Ausdruck auftreten, wenn er mit einem weiteren Operanden verknüpft wird.

Beispiel 2.4.8:

```
var s1, s2: string [6];
    s3 : string [11];
      ⋮
  s1 := 'PASCAL';
  s2 := '–XSC';
  s3 := s1 + s2; {s3 erhält den Wert 'PASCAL–XSC'}
```

2.4.3.3 Record-Ausdruck

PASCAL–XSC

Ein Record-Ausdruck kann selbstdefinierte Operatoren und als Operanden Record-Variablen und Funktionsaufrufe mit Record-Ergebnistyp enthalten. Es gibt keine Standardoperatoren für die Verknüpfung dieser Operanden.

2.4.3.4 Mengen-Ausdruck

Zur Darstellung einer Menge sind Mengenkonstruktoren der Form

[Ausdrucksliste]

vorgesehen, wobei die Ausdrücke vom Elementtyp sein müssen und auch Unterbereiche der Form

Ausdruck..Ausdruck

sein können.

Diese Mengenkonstruktoren sind neben Mengen-Variablen, Funktionsaufrufen und geklammerten Mengenausdrücken als Operanden in einem Mengen-Ausdruck zulässig.

Die Operatoren $+, -, *$ bedeuten Vereinigung, Differenz und Durchschnitt von Mengen.

Beispiel 2.4.9:

```
var mk, mv : set of 'a'..'z';

mv := ['a', 'e', 'i', 'o', 'u']; { Menge der Vokale }
mk := ['a'..'z'] − mv; { Menge der Konsonanten }
```

2.4.3.5 Pointer-Ausdruck

Ein Pointerausdruck besteht aus der Konstanten **nil**, einer Pointer-Variablen oder einem Pointer-Funktionsaufruf. Es gibt keine Standardoperatoren für die Verknüpfung dieser Operanden.

PASCAL–XSC

Die Standardfunktion *ord*, angewandt auf ein Argument vom Pointertyp, liefert den Wert des Zeigers, also die implementierungsabhängige Adresse des Objekts, auf das der Pointer zeigt. Es gilt somit für einen Zeiger *p*:

ord (p) = loc (p↑)

2.4.4 Erweiterte #-Ausdrücke

PASCAL–XSC

Das Konzept der reellen Lattenkreuzausdrücke (#-Ausdrücke) auf der Basis des Datentyps *dotprecision* kann noch erweitert werden für die arithmetischen Standardtypen *complex*, *interval* und *cinterval* mit den in den Arithmetikmodulen (Kapitel 3) vordefinierten Operatoren $+, -, *$. Desweiteren kann man analog Lattenkreuzausdrücke für Vektoren und Matrizen über den Standardtypen *real, interval, complex* und *cinterval* bilden.

PASCAL–XSC

Achtung: Bei der Verwendung dieser höheren #-Ausdrücke muß jeweils das entsprechende Arithmetikmodul, in dem die Operatoren (auch für die Verknüpfung verschiedener Operandentypen) definiert sind, eingebunden werden (siehe Kapitel 3).

Die Operatoren $+, -, *$ haben innerhalb des GEN AUSD die Bedeutung der mathematisch exakten (ungerundeten) Operationen. Sie können deshalb <u>*nicht*</u> durch selbstdefinierte Operatoren überladen werden.

Auch bei den erweiterten #-Ausdrücken kann die Laufanweisung mit **sum** zur abkürzenden Schreibweise von Summen eingesetzt werden. Der GEN AUSD in der Klammer kann dabei wiederum auch von der Laufvariablen i abhängen. Ein Ausdruck der Form

$$\text{GEN AUSD}_a + \text{GEN AUSD}_{a+1} + \ldots + \text{GEN AUSD}_{e-1} + \text{GEN AUSD}_e$$

kann dadurch abgekürzt als

for $i{:=}\ a$ **to** e **sum** (GEN AUSD$_i$)

geschrieben werden (vgl. auch Abschnitt 2.5.8.3 Laufanweisung). Eine leere Laufanweisung entspricht einem Summanden mit Wert Null (für Vektoren und Matrizen komponentenweise). Entsprechendes gilt bei Verwendung von **downto**. In den *integer*-Ausdrücken a und e dürfen dabei nicht selbst wieder explizit #-Ausdrücke auftreten.

2.4.4.1 #-Ausdrücke für die arithmetischen Standardtypen

PASCAL–XSC

Zunächst einmal sind im *real*-Lattenkreuzausdruck als zusätzliche Operanden auch Skalarprodukte von zwei reellen Vektoren, z. B. a, b vom Typ *rvector* in der Form $a * b$ zugelassen. Der Operator $*$ wird im Arithmetikmodul MV_ARI bereitgestellt. Innerhalb eines Lattenkreuzausdrucks bedeutet $a * b$ die exakte Berechnung des Skalarproduktes.

Natürlich sind Lattenkreuzausdrücke auch für die neuen arithmetischen Standardtypen *interval, complex* und *cinterval* interessant und sinnvoll. Syntaktisch werden diese in folgender Form angesprochen:

Der IR LATTENKREUZ AUSDRUCK (Rundung zum kleinsten umfassenden reellen Intervall):

(R GEN AUSD)
(IR GEN AUSD)

PASCAL–XSC

Der CR LATTENKREUZ AUSDRUCK (mit komponentenweiser Rundung zur nächstgelegenen, nächstkleineren oder nächstgrößeren *complex*-Zahl):

| #* (CR GEN AUSD)
| #< (CR GEN AUSD)
| #> (CR GEN AUSD)

Der CIR LATTENKREUZ AUSDRUCK (Rundung zum kleinsten umfassenden komplexen Intervall):

| ## (CR GEN AUSD)
| ## (CIR GEN AUSD)

Die in diesen erweiterten Lattenkreuzausdrücken vorkommenden genau auszuwertenden Ausdrücke (GEN AUSD) haben den analogen syntaktischen Aufbau wie der R GEN AUSD, indem man R durch IR, CR bzw. CIR ersetzt, mit der Ausnahme, daß es keine *interval*- bzw. *complex*- bzw. *cinterval*-DOT VAR gibt.

Allgemein hat ein GENAU AUSZUWERTENDER AUSDRUCK die syntaktische Form

| ± Summand
| Operator Summand ...

wobei als Operator nur + oder − zugelassen ist.

Die zulässigen Summanden, die durch die Operatoren + und − verknüpft werden können, sind (mit $\tau, \sigma \in$ {R, IR, CR, CIR}) die folgenden:

DOT VAR	*dotprecision*-Variable
τ OPD	Konstante, Variable, Funktionsaufruf
τ OPD $*$ σ OPD	exaktes, doppelt langes Produkt zweier Operanden
Vτ OPD $*$ Vσ OPD	exaktes Skalarprodukt zweier Vektoren
(τ GEN AUSD)	geklammerter genau auszuwertender Ausdruck
for i:= a **to** e **sum** (τ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a, e* I Ausdrücke sind
for i:= a **downto** e **sum** (τ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a, e* I Ausdrücke sind

PASCAL–XSC

Nicht alle Summanden müssen vom gleichen Typ sein. Innerhalb eines CIR-Lattenkreuzausdrucks können auch Summanden vom Typ *real, complex* oder *interval* gemischt vorkommen. Dabei ist der Typ des genau auszuwertenden Ausdrucks gemäß der mathematischen Konvention durch die Kombination der einzelnen Summandentypen bestimmt. Die zulässigen τ- bzw. σ-Operanden werden in Abschnitt 2.4.4.4 gesammelt angegeben.

Beispiel 2.4.10:

Mit den Vereinbarungen **var** a, b : real;
ca : complex;
cia : cinterval;
v, w : rvector[1..10];
cv, cw : cvector[1..10];
civ : civector[1..10];

sind die folgenden Lattenkreuzausdrücke syntaktisch möglich

Lattenkreuzausdruck		Ergebnistyp
#<	(b + v * w + **for** i:=1 **to** 10 **sum** (v[i]))	real
#*	(ca + a * b + a * ca + cv * w + cv[3] * cw[5])	complex
##	(b + a * b + v * w)	interval
##	(ca + a * b + ca * cia + cv * civ)	cinterval

2.4.4.2 #-Ausdrücke für Vektoren

PASCAL–XSC

Für Vektoren über den arithmetischen Standardtypen *real, complex, interval* und *cinterval* können entsprechend Lattenkreuzausdrücke (abgekürzt LK AUSD) formuliert werden:

Lattenkreuzausdruck	Ergebnistyp	Syntax
VR LK AUSD	*rvector*	#* (VR GEN AUSD) #< (VR GEN AUSD) #> (VR GEN AUSD)
VIR LK AUSD	*ivector*	## (VR GEN AUSD) ## (VIR GEN AUSD)
VCR LK AUSD	*cvector*	#* (VCR GEN AUSD) #< (VCR GEN AUSD) #> (VCR GEN AUSD)
VCIR LK AUSD	*civector*	## (VCR GEN AUSD) ## (VCIR GEN AUSD)

PASCAL–XSC

Der GENAU AUSZUWERTENDE AUSDRUCK (GEN AUSD) hat wiederum die Form

± Summand			
	Operator	Summand	...

wobei als Operator nur + oder − zugelassen ist.

Die durch + bzw. − verknüpfbaren Summanden haben für $\tau, \sigma \in$ {R, IR, CR, CIR} die Gestalt:

Vτ OPD	Variable, Funktionsaufruf
τ OPD $*$ Vσ OPD	exaktes, doppelt langes Produkt (komponentenweise)
Vτ OPD $*$ σ OPD	exaktes, doppelt langes Produkt (komponentenweise)
Mτ OPD $*$ Vσ OPD	exaktes Matrix–Vektorprodukt (mit komponentenweiser exakter Skalarproduktbildung)
(Vτ GEN AUSD)	geklammerter genau auszuwertender Ausdruck
for i:= a **to** e **sum** (Vτ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a*, *e* I Ausdrücke sind
for i:= a **downto** e **sum** (Vτ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a*, *e* I Ausdrücke sind

Nicht alle Summanden müssen vom gleichen Vektortyp sein. Innerhalb eines VCIR-Lattenkreuzausdrucks können auch Summanden vom Typ *rvector, cvector* oder *ivector* gemischt vorkommen. Dabei ist der Typ des genau auszuwertenden Ausdrucks gemäß der mathematischen Konvention durch die Kombination der einzelnen Summandentypen bestimmt. Die zulässigen Operanden sind im Abschnitt 2.4.4.4 für alle Typen ausführlich aufgelistet.

PASCAL–XSC

Beispiel 2.4.11:

Mit den Vereinbarungen

var	a, b	: real;
	ca	: complex;
	v, w	: rvector[1..10];
	cv	: cvector[1..10];
	iv	: ivector[1..10];
	civ	: civector[1..10];
	M	: rmatrix[1..10,1..10];
	cM	: cmatrix[1..10,1..10];
	iM	: imatrix[1..10,1..10];
	ciM	: cimatrix[1..10,1..10];

sind die folgenden Lattenkreuzausdrücke syntaktisch möglich

Lattenkreuzausdruck		Ergebnistyp
#*	(**for** i:=1 **to** 10 **sum** (M * v + a * M[*,i]))	rvector
#>	(cv + v * b + a * cv + cM * w + cM * cv)	cvector
##	(v + a * v + iM * iv)	ivector
##	(cv + M * v + ca * civ + ciM * cv)	civector

2.4.4.3 #-Ausdrücke für Matrizen

PASCAL–XSC

Entsprechend können Lattenkreuzausdrücke auch für Matrizen über den arithmetischen Standardtypen formuliert werden:

Lattenkreuzausdruck	Ergebnistyp	Syntax
MR LK AUSD	*rmatrix*	#* (MR GEN AUSD) #< (MR GEN AUSD) #> (MR GEN AUSD)
MIR LK AUSD	*imatrix*	## (MR GEN AUSD) ## (MIR GEN AUSD)
MCR LK AUSD	*cmatrix*	#* (MCR GEN AUSD) #< (MCR GEN AUSD) #> (MCR GEN AUSD)
MCIR LK AUSD	*cimatrix*	## (MCR GEN AUSD) ## (MCIR GEN AUSD)

PASCAL–XSC

Der GENAU AUSZUWERTENDE AUSDRUCK (GEN AUSD) hat wiederum die Form

± Summand
Operator Summand ...

wobei als Operator nur + oder − zugelassen ist.

Die durch + bzw. − verknüpfbaren Summanden haben für $\tau, \sigma \in$ {R, IR, CR, CIR} die Gestalt:

Mτ OPD	Variable, Funktionsaufruf
τ OPD $*$ Mσ OPD	exaktes, doppelt langes Produkt (komponentenweise)
Mτ OPD $*$ σ OPD	exaktes, doppelt langes Produkt (komponentenweise)
Mτ OPD $*$ Mσ OPD	exaktes Matrixprodukt (mit komponentenweiser exakter Skalarproduktbildung)
(Mτ GEN AUSD)	geklammerter genau auszuwertender Ausdruck
for i:= a **to** e **sum** (Mτ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a*, *e* I Ausdrücke sind
for i:= a **downto** e **sum** (Mτ GEN AUSD)	Summations-Laufanweisung, wobei *i* eine I Variable und *a*, *e* I Ausdrücke sind

Nicht alle Summanden müssen vom gleichen Matrixtyp sein. Innerhalb eines MCIR-Lattenkreuzausdrucks können auch Summanden vom Typ *rmatrix*, *cmatrix* oder *imatrix* gemischt vorkommen. Dabei ist der Typ des genau auszuwertenden Ausdrucks gemäß der mathematischen Konvention durch die einzelnen Summandentypen bestimmt. Die zulässigen Operanden sind im Abschnitt 2.4.4.4 für alle Typen ausführlich aufgelistet.

Beispiel 2.4.12:

Mit den Vereinbarungen aus Beispiel 2.4.11 sind die folgenden #-Ausdrücke syntaktisch möglich:

Lattenkreuzausdruck		Ergebnistyp
#>	(**for** i:=1 **to** 10 **sum** (M $*$ M))	rmatrix
#<	(cM + b $*$ cM + **for** i:=1 **to** 10 **sum** (cM $*$ M))	cmatrix
##	(M + a $*$ M + iM $*$ iM)	imatrix
##	(cM + M $*$ iM + ca $*$ iM + cM $*$ cM)	cimatrix

2.4.4.4 Liste der Operanden im #-Ausdruck

PASCAL–XSC

Die genaue Funktionsweise der aufgeführten Funktionen, ihre Schnittstellen und Vereinbarungen sind im Kapitel 3 (Arithmetikmodule) und in den Verzeichnissen im Anhang beschrieben.

R OPD:
I VAR
I KONST
R VAR
R KONST
inf (IR OPD) (untere Intervallgrenze)
sup (IR OPD) (obere Intervallgrenze)
re (CR OPD) (Realteil)
im (CR OPD) (Imaginärteil)

IR OPD:
IR VAR
intval (R OPD) (Punktintervall)
intval (R OPD, R OPD) (Intervallbildung)
re (CIR OPD) (Realteil)
im (CIR OPD) (Imaginärteil)

CR OPD:
CR VAR (z. B. $z = x + iy$)
conj (CR OPD) (Konjugation $\overline{z} = x - iy$)
compl (R OPD) (rein reelle *complex*-Zahl)
compl (R OPD, R OPD) (Komplexbildung)
inf (CIR OPD) (untere Intervallgrenze)
sup (CIR OPD) (obere Intervallgrenze)

CIR OPD:
CIR VAR
conj (CIR OPD)
intval (CR OPD)
intval (CR OPD, CR OPD)
intval (R OPD, CR OPD)
intval (CR OPD, R OPD)
compl (IR OPD)
compl (IR OPD, IR OPD)
compl (R OPD, IR OPD)
compl (IR OPD, R OPD)

PASCAL–XSC

VR OPD:	VR VAR
	rvector (A VAR) (Qualifikation eines gleichstruktur. Feldes)
	inf (VIR OPD)
	sup (VIR OPD)
	re (VCR OPD)
	im (VCR OPD)
VIR OPD:	VIR VAR
	ivector (A VAR) (Qualifikation eines gleichstruktur. Feldes)
	intval (VR OPD)
	intval (VR OPD, VR OPD)
	re (VCIR OPD)
	im (VCIR OPD)
VCR OPD:	VCR VAR
	cvector (A VAR) (Qualifikation eines gleichstruktur. Feldes)
	conj (VCR OPD)
	compl (VR OPD)
	compl (VR OPD, VR OPD)
	inf (VCIR OPD)
	sup (VCIR OPD)
VCIR OPD:	VCIR VAR
	civector (A VAR) (Qualifikation eines gleichstruktur. Feldes)
	conj (VCIR OPD)
	intval (VCR OPD)
	intval (VCR OPD, VCR OPD)
	intval (VR OPD, VCR OPD)
	intval (VCR OPD, VR OPD)
	compl (VIR OPD)
	compl (VIR OPD, VIR OPD)
	compl (VR OPD, VIR OPD)
	compl (VIR OPD, VR OPD)
MR OPD:	MR VAR
	rmatrix (A VAR) (Qualifikation eines gleichstruktur. Feldes)
	id (MR OPD) (Einheitsmatrix)
	id (MR OPD, MR OPD) (Einheitsmatrix)
	transp (MR OPD) (Transponierte Matrix)
	inf (MIR OPD)
	sup (MIR OPD)
	re (MCR OPD)
	im (MCR OPD)

PASCAL–XSC

MIR OPD: MIR VAR
imatrix (A VAR) (Qualifikation eines gleichstruktur. Feldes)
id (MIR OPD)
id (MIR OPD, MIR OPD)
transp (MIR OPD)
intval (MR OPD)
intval (MR OPD, MR OPD)
re (MCIR OPD)
im (MCIR OPD)

MCR OPD: MCR VAR
cmatrix (A VAR) (Qualifikation eines gleichstruktur. Feldes)
id (MCR OPD)
id (MCR OPD, MCR OPD)
transp (MCR OPD)
herm (MCR OPD) (Hermitesche Matrix)
conj (MCR OPD)
compl (MR OPD)
compl (MR OPD, MR OPD)
inf (MCIR OPD)
sup (MCIR OPD)

MCIR OPD: MCIR VAR
cimatrix (A VAR) (Qualifikation eines gleichstruktur. Feldes)
id (MCIR OPD)
id (MCIR OPD, MCIR OPD)
transp (MCIR OPD)
herm (MCIR OPD)
conj (MCIR OPD)
intval (MCR OPD)
intval (MCR OPD, MCR OPD)
intval (MR OPD, MCR OPD)
intval (MCR OPD, MR OPD)
compl (MIR OPD)
compl (MIR OPD, MIR OPD)
compl (MR OPD, MIR OPD)
compl (MIR OPD, MR OPD)

2.4.4.5 Übersicht über die allgemeinen #-Ausdrücke

PASCAL–XSC

Die folgenden Tabellen geben einen Gesamtüberblick über die verschiedenen Verwendungsmöglichkeiten des #-Ausdrucks.

Reelle und komplexe #-Ausdrücke

Syntax: #-Symbol (GEN AUSD)

#-Symbol	Ergebnistyp	Erlaubte Summanden im GEN AUSD
#	*dotprecision*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real* oder *dotprecision* • Produkte vom Typ *integer* oder *real* • Skalarprodukte vom Typ *real*
#* #< #>	*real*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real* oder *dotprecision* • Produkte vom Typ *integer* oder *real* • Skalarprodukte vom Typ *real*
	complex	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *complex* oder *dotprecision* • Produkte vom Typ *integer*, *real* oder *complex* • Skalarprodukte von Typ *real* oder *complex*
	rvector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* • Produkte vom Typ *rvector* (z. B. *rmatrix* ∗ *rvector*, *real* ∗ *rvector* etc.)
	cvector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* oder *cvector* • Produkte vom Typ *rvector* oder *cvector* (z. B. *cmatrix* ∗ *rvector*, *real* ∗ *cvector* etc.)
	rmatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* • Produkte vom Typ *rmatrix*
	cmatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* oder *cmatrix* • Produkte vom Typ *rmatrix* oder *cmatrix*

PASCAL–XSC

Reelle und komplexe Intervall-#-Ausdrücke

Syntax: ## (GEN AUSD)

#-Symbol	Ergebnistyp	Erlaubte Summanden im GEN AUSD
##	*interval*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *interval* oder *dotprecision* • Produkte vom Typ *integer*, *real* oder *interval* • Skalarprodukte vom Typ *real* oder *interval*
	cinterval	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *complex*, *interval*, *cinterval* oder *dotprecision* • Produkte vom Typ *integer*, *real*, *complex*, *interval* oder *cinterval* • Skalarprodukte vom Typ *real*, *complex*, *interval* oder *cinterval*
	ivector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* oder *ivector* • Produkte vom Typ *rvector* oder *ivector*
	civector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector*, *cvector*, *ivector* oder *civector* • Produkte vom Typ *rvector*, *cvector*, *ivector* oder *civector*
	imatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* oder *imatrix* • Produkte vom Typ *rmatrix* oder *imatrix*
	cimatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix*, *cmatrix*, *imatrix* oder *cimatrix* • Produkte vom Typ *rmatrix*, *cmatrix*, *imatrix* oder *cimatrix*

2.5 Anweisungen

In PASCAL gibt es einfache und strukturierte Anweisungen. Zu den einfachen Anweisungen gehören die Wertzuweisung, die Eingabe- bzw. Ausgabeanweisung, die leere Anweisung, die Prozeduranweisung und die **goto**-Anweisung. Die Verbundanweisung, die bedingten Anweisungen, die Wiederholungsanweisungen und die **with**-Anweisung zählen zu den strukturierten Anweisungen.

2.5.1 Wertzuweisung

Durch eine Wertzuweisung wird einer Variablen der Wert eines Ausdrucks zugewiesen:

Variable := Ausdruck

Der Typ des rechtsstehenden Ausdrucks muß mit dem Typ der linksstehenden Variablen zuweisungsverträglich sein (vgl. Abschnitt 2.3.5). Zunächst wird der Ausdruck ausgewertet und anschließend dieser Wert der Variable zugewiesen, d. h. in dem entsprechenden Speicherbereich abgespeichert. Die Reihenfolge des Zugriffs auf die Variable auf der linken Seite und der Auswertung des Ausdrucks auf der rechten Seite der Zuweisung ist implementierungsabhängig.

Innerhalb einer Funktionsprozedur muß die Zuweisung des Ergebniswertes an den Funktionsnamen erfolgen. Dazu darf der Funktionsname wie eine Variable vom Funktionstyp auf der linken Seite einer Wertzuweisung auftreten.

Beispiel 2.5.1:

```
var
  r, x : real;
  i, k : integer;
...
i := k div 3 + 1;
r := i div k;
r := x * x + sin (x);

i := r * x; { !! nicht zulässig !! }
```

PASCAL–XSC

In PASCAL–XSC ist es möglich, durch Überladung des Zuweisungsoperators := die Wertzuweisung auch für nicht verträgliche Typen zu definieren, indem man in einem Unterprogramm den Algorithmus bzw. die Vorgehensweise für eine solche Zuweisung festlegt (vgl. Abschnitt 2.7.12).

2.5.2 Eingabe- bzw. Ausgabeanweisungen

Mit Hilfe der Ein- bzw. Ausgabeanweisungen *read, readln, write* und *writeln* können Dateien (Filevariable) zur Ein- bzw. Ausgabe benutzt werden. Dabei kann es sich um allgemeine Dateien vom Typ **file of ...**, um Textdateien vom Typ *text* oder um die Standarddateien *input* und *output* handeln. Fehlt in der entsprechenden Anweisung die Angabe eines Files, so ist beim Lesen (Eingabe) automatisch die Standarddatei *input* und beim Schreiben (Ausgabe) die Standarddatei *output* angesprochen. In diesem Fall muß im Programmkopf *input* bzw. *output* als Programmparameter angegeben werden (vgl. Abschnitt 2.6). Sollen den PASCAL-Files Dateien der Systemumgebung zugeordnet werden, so müssen auch diese Files in der Programmparameterliste aufgeführt werden.

Eröffnen von Dateien

Die Standard-Textdateien *input* und *output* werden bei Bedarf automatisch eröffnet. Alle anderen Dateien müssen explizit eröffnet werden.

reset (t)
: Die Datei *t* wird zum Lesen eröffnet. Nach *reset(t)* enthält *t*↑ das erste Element der Datei. Falls dieses nicht vorhanden ist, ist *eof(t)* = *true*. Nach dem beim Programmstart automatisch durchgeführten *reset(input)* hat *eoln(input)* den Wert *true* und in *input*↑ steht dementsprechend ein Leerzeichen.

rewrite (t)
: Die Datei *t* wird zum Schreiben eröffnet. *t* zeigt auf die erste aktuelle Position, die zum Beschreiben genutzt werden kann. Der Wert von *t*↑ ist undefiniert.

Eingabeanweisungen

read (t, v1, ..., vn)
: Von der Datei *t* werden Werte für die Variablen *v1, ..., vn* in dieser Reihenfolge gelesen. Diese Anweisung entspricht den Anweisungen

    read (t, v1); ... read (t, vn);

  Jedes *read (t, v)* ist für den allgemeinen Filetyp (*file* **of** ...) definiert durch

    begin v := t↑; get (t); **end**

  Dies gilt auch für Textdateien (*text*), wenn *v* eine Variable vom Typ *char* ist, während für *integer*- oder *real*-Variablen eine ganze Folge von Zeichen von der Textdatei gelesen und zu einer entsprechenden Zahl verarbeitet wird. Die Folge von Zeichen muß dabei der in Abschnitt 2.3.1 beschriebenen Syntax von Literalkonstanten entsprechen. Führende Leerzeichen oder Zeilenendezeichen werden ignoriert, und der Lesevorgang ist beendet, wenn *t*↑ kein

Teil der zu lesenden Zahl sein kann (vgl. auch Abschnitt 2.9).

readln (t) bzw. readln — Die restlichen Zeichen der laufenden Zeile werden gelesen, und es wird der Übergang an den Anfang der nächsten Zeile vorgenommen (nur für Textfiles!). *readln (t)* ist definiert durch

```
begin
   while not eoln (t) do
      get (t);
   get (t)
end;
```

readln (t, v1, ..., vn) — entspricht der zusammengesetzten Anweisung

```
begin
   read (t, v1, ..., vn);
   readln (t)
end;
```

Ausgabeanweisungen

write (t, a1, ..., an) — Es werden die Werte der Ausdrücke *a1, a2, ..., an* berechnet und in dieser Reihenfolge auf die Datei *t* ausgegeben. Diese Anweisung entspricht den Anweisungen

write (t, a1); ... write (t, an);

Jedes *write (t, a)* ist für den allgemeinen Filetyp (*file* **of** ...) definiert durch

begin t↑:= a; put (t); **end**

Dies gilt auch für Textdateien (*text*), wenn *a* ein Ausdruck vom Typ *char* ist, während für *integer*-, *real*- oder *boolean*-Ausdrücke eine ganze Folge von Zeichen, die den entsprechenden Wert darstellt, in einem Standardformat und in der notwendigen Anzahl von Zeilen auf die Textdatei ausgegeben wird.

Bei *integer*-Werten erfolgt die Darstellung als Dezimalzahl ohne führende Nullen. Vorzeichen werden nur für negative Werte ausgegeben. Bei *real*-Werten erfolgt die Darstellung als dezimale Gleitpunktzahl mit einer signifikanten Ziffer vor dem Dezimalpunkt und vorangehendem Minuszeichen für negative Werte, Leerzeichen für nichtnegative Werte und einem Exponententeil mit führendem Zeichen *E*.

	Die logischen Werte werden als *true* bzw. *false* ausgegeben. Bei *char*-Werten ist das Zeichen selbst (ohne Hochkomma) die Ausgabe. Bei einem Zeichenkettenwert wird die Sequenz der enthaltenen Zeichen mit der benötigten Anzahl von Druckpositionen ausgegeben.
writeln (t) bzw. writeln	Die laufende Zeile wird abgeschlossen. Die nächste Ausgabe erfolgt vom Beginn der nächstfolgenden Zeile (nur für Textfiles!). *writeln (t)* ist definiert durch

```
begin
   t↑ := „Zeilenendezeichen“;
   put (t)
end;
```

writeln (t, a1, ..., an)	entspricht der zusammengesetzten Anweisung

```
begin
   write (t, a1, ..., an);
   writeln(t)
end;
```

page (t)	Alle nachfolgenden Ausgaben erfolgen auf einer neuen „Seite“ (nur für Textfiles, implementierungsabhängig!).

Formatspezifikation

Wird beim Schreiben auf Textdateien nach einem Ausgabeargument *a* ein Doppelpunkt mit anschließendem *integer*-Ausdruck (Steuerausdruck) $p > 0$ in der Form

```
write (a : p);
```

angegeben, so legt der Wert dieses Ausdrucks *p* die minimale Anzahl der Druckpositionen für das Ausgabeargument fest.

Für Ausgabeargumente vom Typ *real* kann ein zweiter Doppelpunkt mit Steuerausdruck *n* vom Typ *integer* angegeben werden:

```
write (a : p : n);
```

Dieser Ausdruck induziert eine Festpunktdarstellung mit der durch den Wert $n > 0$ gegebenen Anzahl von Ziffern im Dezimalbruch (Nachkommastellen).

PASCAL–XSC

Die Prozeduren *reset* und *rewrite* können mit einem zweiten Parameter *s* vom Typ *string* aufgerufen werden gemäß

```
reset (t, s)
rewrite (t, s)
```

Dadurch wird der Datei *t* der physikalische Dateiname *s* zugeordnet.

Das in PASCAL–XSC verfügbare Überladungsprinzip (Abschnitt 2.7.10) gilt auch für die Standardprozeduren *read* und *write*. Dies ermöglicht eine Verwendung dieser Standardprozeduren mit einer beliebigen Anzahl von Argumenten und Formatsteuerungen auch für benutzerdefinierte Typen (vgl. Abschnitt 2.7.11).

Für die Ein- und Ausgabe von *real*-Werten stellt PASCAL–XSC die Prozeduren *read* und *write* (bzw. *readln* und *writeln*) mit einem zusätzlichen Formatsteuerparameter *r* zur Verfügung. Dieser *integer*-Ausdruck spezifiziert die Rundung, mit der eine *real*-Zahl ein- bzw. ausgegeben wird.

Die Eingabe eines Wertes vom Typ *real* kann auch mit mehr Ziffern erfolgen, als die Mantissenlänge der internen Darstellung beträgt. Unter Verwendung der Anweisung

```
read (v : r)
```

wird die Eingabe entsprechend dem Wert des Rundungsparameters *r* auf die interne Darstellung gerundet. Entsprechend kann die Ausgabe des Wertes eines *real*-Ausdrucks *a* durch Verwendung einer Anweisung der Form

```
write (a : p : n : r)
```

auf *n* Nachkommastellen gerundet erfolgen. In beiden Fällen hat der Parameter *r* folgende Bedeutung:

r	Rundungsmodus
fehlt	zur nächstgelegenen darstellbaren Zahl
0	zur nächstgelegenen darstellbaren Zahl
< 0	zur nächstkleineren darstellbaren Zahl
> 0	zur nächstgrößeren darstellbaren Zahl

Auch bei der Wandlung von *real*-Werten in Strings kann ein solcher Rundungsparameter verwendet werden (vgl. Abschnitt 2.9).

Um *real*-Werte auch bei Verwendung des Rundungsparameters im Gleitkommaformat ausgeben zu können, ist $n = 0$ als zweiter Formatsteuerparameter erlaubt. Außerdem ist in diesem Fall auch $p = 0$ als erster Formatsteuerparameter zulässig, um damit das Standard-Gleitkommaformat anzusprechen.

PASCAL–XSC

Die Verwendung des Rundungsparameters empfiehlt sich bei der Ausgabe von Werten, die mittels gerichteter Operationen erzeugt wurden, um die je nach Implementierung notwendige Konvertierung ins dezimale Ausgabeformat ebenfalls gerichtet zu erfassen.

Beispiel 2.5.2:

```
var x: real; { 14 dezimale Mantissenstellen }
begin
    read (x : +1);
    writeln (x : 11 : 0 : −1, ' ', x : 9 : 3 : 1);
end.
```

Eingabe:	`␣4730281356200104E-12`
Wert von x:	`4.7302813562002E3`
Ausgabe:	`␣4.7302E+03␣␣4730.282`

Für die zusätzlichen numerischen Standardtypen sind die Überladungen von *read* und *write* in den Arithmetikmodulen vordefiniert (siehe Kapitel 3). Bei neuen, selbstdefinierten Datentypen kann die Überladung zusätzlich explizit vereinbart werden (siehe Abschnitt 2.7.11).

2.5.3 Leere Anweisung

Die leere Anweisung kann überall dort verwendet werden, wo syntaktisch eine Anweisung erforderlich ist, eine Aktion vom Programmierer jedoch nicht beabsichtigt ist. Da für die leere Anweisung keine Symbole vorgesehen sind, ist sie nur am Kontext erkennbar, etwa zwischen den Symbolen

; ;   oder   ; **end**   oder   **then else** usw.

Sinnvoll wird die leere Anweisung im Zusammenhang mit der **goto**-Anweisung beim Sprung an das Ende eines Blocks verwendet.

Beispiel 2.5.3:

```
goto 100;
  ⋮
100: { leere Anweisung }
end;
```

2.5.4 Prozeduranweisung

Eine Prozeduranweisung bewirkt den Aufruf der genannten Prozedur mit den aktuellen Argumenten an Stelle der formalen Argumente:

Prozedurname
(Aktuelle Argumentliste) { kann entfallen }

Ist die Prozedur ohne formale Argumente vereinbart worden, so muß auch der Aufruf ohne aktuelle Argumentliste erfolgen. Andernfalls müssen die aktuellen Argumente in der gegebenen Reihenfolge zu den formalen Argumenten passen, d. h. bei Referenzaufruf müssen sie Variable vom kompatiblen Typ sein, bei Wertaufruf sind Ausdrücke mit zuweisungskompatiblem Wert erforderlich. Weitere Details sind in Abschnitt 2.7.1 zu finden.

PASCAL–XSC

PASCAL–XSC erlaubt einen modifizierten Referenzaufruf in Verbindung mit strukturierten Datentypen (siehe Abschnitt 2.7.9).

Beispiel 2.5.4:

quicksort (x, i, j); { Aufruf einer Sortierprozedur }
primzahl (m); { Aufruf einer Prozedur zur Primzahlenberechnung }

2.5.5 goto-Anweisung

Die **goto**-Anweisung (Sprunganweisung) dient zum Springen an eine andere, durch eine Marke gekennzeichnete Anweisung. Der sequentielle Ablauf des Programms wird dabei unterbrochen und mit der Ausführung der markierten Anweisung fortgesetzt. Die **goto**-Anweisung hat die Form:

goto Marke

Dabei ist Marke eine vorzeichenlose ganze Zahl mit maximal vier Ziffern, die die anzuspringende Anweisung in der Form

Marke : Anweisung

kennzeichnet.

Alle Marken, die Anweisungen markieren, sind im Markenvereinbarungsteil des entsprechenden Blocks zu vereinbaren durch:

label
Markenliste ; { nicht leer }

Eine **goto**-Anweisung darf nur zu einer Anweisung springen, die auf gleichem bzw. höherem Niveau bezüglich der Blockstruktur steht. Es darf jedoch nicht von außerhalb in das Innere einer strukturierten Anweisung gesprungen werden.

Die Verwendung der Sprunganweisung ist wegen der zu beachtenden Blockstruktur, der Unsicherheiten bei strukturierten Anweisungen und der Gefahr von unübersichtlichem Programmablauf möglichst zu vermeiden.

2.5.6 Verbundanweisung

Eine Verbundanweisung faßt eine Sequenz von Anweisungen zu einer einzigen Anweisung zusammen:

```
begin
  Anweisung; ...
end
```

Die Ausführung einer Verbundanweisung erfolgt analog zur Ausführung des Anweisungsteiles eines Programmes.

Beispiel 2.5.5:

```
while i <= n do
begin
   s := s + a[i];
   i := i + 1
end;
```

2.5.7 Bedingte Anweisungen

2.5.7.1 if-Anweisung

Die **if**-Anweisung ermöglicht die wahlweise Ausführung von zwei Anweisungen:

```
if  logischer Ausdruck  then  Anweisung
                        else  Anweisung   { kann entfallen }
```

Die Ausführung der bedingten Anweisung bewirkt die Auswertung des logischen Ausdrucks. Ist der Wert des Ausdrucks *true*, so wird die nach **then** folgende Anweisung (1. Alternative) ausgeführt, andernfalls die nach **else** folgende Anweisung (2. Alternative). Danach ist die Ausführung der bedingten Anweisung beendet. Man beachte, daß der **else**-Zweig fehlen darf. Dieser Fall wird behandelt, als wäre dieser Zweig mit der leeren Anweisung vorhanden.

Beispiel 2.5.6:

```
if x <= y   then  z := y − x
            else  z := x − y; {positive Differenz von x und y}
if x >= 0   then  y := sqrt (x);
```

2.5.7.2 case-Anweisung

Im Unterschied zur **if**-Anweisung erlaubt die **case**-Anweisung oder auch Auswahlanweisung die Ausführung einer Anweisung, welche aus beliebig vielen Alternativen ausgewählt wird:

```
case  Auswahlausdruck of
      Auswahlkonstantenliste: Anweisung; ...  { nicht leer }
end
```

Bei Ausführung einer Auswahlanweisung wird zunächst der Auswahlausdruck ausgewertet. Ist der Wert in einer Auswahlkonstantenliste enthalten, so wird die nachfolgende Anweisung ausgeführt. Falls der Wert nicht in einer Auswahlkonstantenliste enthalten ist, erfolgt eine Fehlermeldung.

Der Auswahlausdruck darf vom Typ *integer, boolean, char* oder Aufzählungstyp sein, die Auswahlkonstanten müssen alle demselben Typ angehören. Im Sinne der Ordnung des zugrundeliegenden Typs können unmittelbar aufeinanderfolgende Auswahlmarken der Auswahlkonstantenliste abkürzend auch in Form eines Unterbereichs

Auswahlkonstante .. Auswahlkonstante

angegeben werden. Verschiedene Auswahlkonstantenlisten müssen disjunkt sein.

PASCAL–XSC

Die **case**-Anweisung kann unmittelbar vor **end** noch einen **else**-Zweig enthalten, der alle nicht aufgelisteten Konstanten des Auswahltyps zusammenfaßt:

else: Anweisung

Dieser **else**-Zweig wird ausgeführt, wenn der Wert des Auswahlausdrucks in keiner der Auswahlkonstantenlisten auftritt.

Beispiel 2.5.7:

```
case trunc (phi/90) + 1 of
    1: f := phi * r;
    2: f := 90 * r;
    3: f := −(phi − 180) * r;
    4: f := − 90 * r;
    else f := 0;
end;
```

2.5.8 Wiederholungsanweisungen

2.5.8.1 while-Anweisung

Die **while**-Anweisung erlaubt die wiederholte Ausführung einer Anweisung unter Kontrolle einer Anfangsbedingung:

while logischer Ausdruck **do** Anweisung

Die auf **do** folgende Anweisung wird solange wiederholt, wie der logische Ausdruck den Wert *true* besitzt. Vor jeder Ausführung der Anweisung wird also der logische Ausdruck ausgewertet. Tritt der Wert *false* auf, ist die Ausführung der **while**-Anweisung beendet. Dies darf auch schon bei der ersten Auswertung der Fall sein.

Beispiel 2.5.8:

```
i := n;
while i >= 1 do
begin
   s := s + a[i];
   i := i - 2
end;
```

2.5.8.2 repeat-Anweisung

Die **repeat**-Anweisung bewirkt die wiederholte Ausführung einer Folge von Anweisungen bis zum Eintreten einer Endbedingung:

```
repeat
   Anweisung; ...
until logischer Ausdruck
```

Die zwischen **repeat** und **until** stehenden Anweisungen werden solange wiederholt ausgeführt, bis der logische Ausdruck den Wert *true* besitzt. Es wird also nach jeder Ausführung der Anweisungsfolge der logische Ausdruck ausgewertet. Dies bedeutet in jedem Fall mindestens eine Ausführung dieser Sequenz.

Beispiel 2.5.9:

```
i := n;
repeat
   s := s + a[i];
   i := i - 2
until i < 1
```

Man beachte, daß unabhängig vom Wert für n die beiden Anweisungen zwischen **repeat** und **until** mindestens einmal ausgeführt werden, also für $n = 0$ z. B. $s := s + a[0];$ und $i := -2$.

2.5.8.3 for-Anweisung

Die **for**-Anweisung oder auch Laufanweisung erlaubt die wiederholte Ausführung einer Anweisung für eine zu Beginn bekannte Anzahl von Wiederholungen:

> **for** Laufvariable := Anfangswert **to** Endwert **do**
>   Anweisung

bzw.

> **for** Laufvariable := Anfangswert **downto** Endwert **do**
>   Anweisung

Es werden zunächst die Ausdücke für den Anfangs- und den Endwert ausgewertet. Ist der Endwert kleiner bzw. größer als der Anfangswert, so ist die Ausführung der Laufanweisung beendet. In diesem Fall spricht man von einer *leeren Laufanweisung*. Andernfalls erhält die Laufvariable den Anfangswert zugewiesen und die Anweisung wird ausgeführt. Ist die Laufvariable ungleich dem Endwert, so wird der Laufvariablen jeweils der Nachfolger- bzw. Vorgängerwert zugewiesen und die Anweisung ausgeführt, bis für den Endwert die letzte Ausführung erfolgt ist. Die Laufvariable darf vom Typ *integer, boolean, char* oder Aufzählungstyp sein und muß in dem Block vereinbart sein, in dem die Laufanweisung steht. Anfangs- und Endwert müssen zuweisungskompatibel sein.

Man beachte, daß die Laufvariable innerhalb der nach **do** folgenden Anweisung *nicht*

- auf der linken Seite einer Wertzuweisung,
- als Aktualparameter zu einem formalen **var**-Parameter bei einem Unterprogrammaufruf,
- als Eingabeparameter innerhalb einer *read*-Anweisung oder
- als Laufvariable einer weiteren **for**-Schleife

auftreten darf.

Beispiel 2.5.10:

for i := 1 **to** n **do** s := s + a[i];
for i := n **downto** 1 **do** s := s + a[i];

Die Variation der Schrittweite kann nur durch explizite Programmierung erfolgen.

Beispiel 2.5.11:

for i := 1 **to** n **do** s := s + a[2∗i];

2.5.9 with-Anweisung

Die **with**-Anweisung erleichtert das Arbeiten mit Record-Komponenten ganz wesentlich, indem sie eine abkürzende Schreibweise für Record-Komponenten gestattet. Sie hat die Form:

> **with** Recordvariablenliste **do** Anweisung

Enthält die Liste nach dem Wortsymbol **with** mehr als eine Recordvariable, also etwa

with r1, r2, ..., rn **do** Anweisung;

so entspricht ihre Wirkung der geschachtelten **with**-Anweisung

with r1 **do with** r2, ..., rn **do** Anweisung;

Es genügt also, die Wirkung der **with**-Anweisung

with r **do** Anweisung;

festzulegen. Diese besteht in der Ausführung der nach **do** stehenden Anweisung, wobei dort Komponenten der Recordvariablen *r* ohne das Präfix *r*. auftreten dürfen.

Beispiel 2.5.12:

```
type datum = record tag: 1 ... 31;
                    monat: (jan, feb, mar, apr, mai, jun,
                            jul, aug, sep, okt, nov, dez);
                    jahr: integer;
             end;
var d: datum;
  ...
begin
  ...
  with d do
  begin
     tag := 14;
     monat := dez;
     jahr := 1990
  end;
  ...
end.
```

2.6 Programmstruktur

Ein Programm besteht aus einem Programmkopf, welcher nach dem einführenden Wortsymbol **program** den Namen des Programms und als Parameter in Klammern die verwendeten externen Dateien (insbesondere *input*, *output*) enthält, aus den notwendigen Vereinbarungen und aus dem in **begin** und **end** eingeschlossenen Anweisungsteil:

```
program Name
   (Programmparameterliste)    { kann entfallen }
   ;
   Vereinbarung; ...
begin
   Anweisung; ...
end.
```

Der Anweisungsteil beschreibt die Bearbeitungsvorschrift (Algorithmus), welche durch den Computer ausgeführt werden soll. Alle in diesem Teil auftretenden Objekte, die nicht vordefiniert, also keine Standardobjekte sind, müssen in einer Vereinbarung erläutert werden. Dabei ist in Standard-PASCAL die Reihenfolge der Vereinbarungsteile wie folgt fest vorgegeben: Markenvereinbarungsteil, Konstantendefinitionsteil, Typendefinitionsteil, Variablenvereinbarungsteil und zuletzt der Prozeduren- und Funktionenvereinbarungsteil.

Bei der Kodierung eines Programms sind folgende Vorschriften für die Verwendung von Trennzeichen zu beachten:

- Innerhalb von Namen, Zahlen und Wortsymbolen sowie zusammengesetzten Zeichensymbolen (z. B. <=, :=) darf kein Trennzeichen stehen.

- Unmittelbar aufeinanderfolgende Namen, Zahlen oder Wortsymbole müssen durch mindestens ein Trennzeichen getrennt werden.

Trennzeichen ist das Leerzeichen (␣). Der Übergang zu einer neuen Zeile oder das Einschieben eines Kommentars, welcher aus einer beliebigen in geschweiften Klammern „{“ , „}“ eingeschlossenen Zeichenfolge bestehen darf, bewirken ebenfalls wie das Leerzeichen die Trennung.

Die Ausführung eines Programms bewirkt die Abarbeitung der Vereinbarungen in der angegebenen Reihenfolge und anschließend die Ausführung des Anweisungsteiles beginnend mit der physikalisch ersten Anweisung. Nach jeder Anweisung wird deren Nachfolgeanweisung ausgeführt. In der Regel ist dies die physikalisch nachfolgende Anweisung. Bei der Sprunganweisung und innerhalb von strukturierten Anweisungen gelten spezielle Festlegungen.

PASCAL–XSC

Ein ausführbares Programm besteht aus einem Hauptprogramm, entsprechend dem Programm in Standard-PASCAL, und eventuell aus einer Anzahl von Modulen, welche entweder im Hauptprogramm selbst in einer **use**-Klausel aufgezählt sind oder in diesen aufgezählten Modulen benutzt werden. Im Hauptprogramm können also Objekte auftreten, die

- vordefiniert sind (Standardobjekte) oder
- im Hauptprogramm definiert bzw. vereinbart werden oder
- in einem benutzten Modul mit **global** definiert sind.

Ein Hauptprogramm hat die Form:

```
program Name
   (Programmparameterliste)    { kann entfallen }
   ;
   Use-Klausel; ...
   Vereinbarung; ...
begin
   Anweisung; ...
end.
```

Die Ausführung eines Programmes bewirkt zunächst die Abarbeitung der zugehörigen Module entsprechend der Modulhierarchie einschließlich der Ausführung der Anweisungsteile, anschließend die Abarbeitung der Vereinbarungen des Hauptprogramms und die Ausführung des Anweisungsteiles wie bei Programmen in Standard-PASCAL.

Die Vereinbarungen, eingeleitet mit **label**, **const**, **type**, **var**, **function**, **procedure**, **priority** oder **operator**, können in beliebiger Reihenfolge und auch jeweils mehrfach auftreten. Dabei muß allerdings beachtet werden, daß ein Name vor seiner weiteren Verwendung vereinbart bzw. definiert sein muß.

2.7 Unterprogramme

Teilalgorithmen werden in Standard-PASCAL als Prozeduren oder Funktionen vereinbart und aufgerufen. Während eine Funktion dazu gedacht ist, bei einer Ausführung ein einziges Ergebnis vom Typ *integer, real, boolean, char*, Aufzählungstyp oder *pointer* zu berechnen, kann in einer Prozedur ein Algorithmus mit beliebiger Anzahl von Ergebnisparametern, deren Typen ebenfalls beliebig sind, realisiert werden. Die Vereinbarung von Prozeduren und Funktionen erfolgt unmittelbar vor dem Anweisungsteil eines Programmes.

PASCAL–XSC

Als weitere Art, Unterprogramme zu realisieren, gibt es in PASCAL–XSC die Möglichkeit, Operatoren zu vereinbaren, deren Ergebnis, wie auch das von Funktionen, von beliebigem Typ sein kann. Die Vereinbarung von Prozeduren, Funktionen und Operatoren kann an beliebiger Stelle des Vereinbarungsteils von Programmen erfolgen.

2.7.1 Prozeduren

Die Form der Prozedurvereinbarung ähnelt derjenigen eines Programmes:

```
procedure Name
   (Formale Argumentliste) { kann entfallen }
   ;
   Vereinbarung; ...
begin
   Anweisung; ...
end;
```

Dabei beschreibt die formale Argumentliste diejenigen Objekte der Prozedur, welche bei deren Aufruf als Ein- und Ausgabeargumente fungieren. Es können formale Argumente für Variablen, Prozeduren und Funktionen verwendet werden.

Die Spezifikation von formalen Argumenten erfolgt durch

```
var    { kann entfallen }
   Namensliste : Typspezifikation
```

Steht das Wortsymbol **var** vor einem Namen oder einer Namensliste, so werden beim Aufruf der Prozedur die aufgezählten Argumente mit Referenzaufruf behandelt, andernfalls mit Wertaufruf.

Die Spezifikation von Prozeduren und Funktionen erfolgt durch Angabe eines entsprechenden Prozedur- oder Funktionskopfes, also inklusive einer formalen Argumentliste und eventuell des Funktionstyps.

Die einzelnen Abschnitte in der formalen Argumentliste werden jeweils durch ein Semikolon (;) getrennt. Es existiert keine Beschränkung für die Länge und die Zusammensetzung der Liste.

Im Gegensatz zu Vereinbarungen darf die Typspezifikation für formale Argumente ein sogenanntes Konform-Array-Schema enthalten:

array [Indexbereichsliste] **of** Typspezifikation

mit Indexbereichen der Form

Name..Name : Typ

und dem Trennzeichen ; in der Liste. Ein Konform-Array-Schema läßt die Indexbereiche des formalen Arguments variabel. Der Zugriff auf die Indexgrenzen erfolgt innerhalb der Prozedur durch die im Schema spezifizierten Namen, die wie formale Argumente verwendet werden können.

Der Anweisungsteil der Prozedur enthält die Bearbeitungsvorschrift für den implementierten Algorithmus. Diese Vorschrift kann formuliert werden unter Benutzung der

- formalen Argumente
- lokalen Objekte der Prozedur (d. h. in der Prozedur vereinbarten Größen)
- globalen Objekte der Prozedur (d. h. Objekte des umfassenden Programmes bzw. der umfassenden Prozedur).

Beispiel 2.7.1:

```
type
   bruch = record Z, N : integer end;

procedure readbruch (var b: bruch);
   begin
      write ('Zaehler = '); read (b.Z);
      write ('Nenner = '); read (b.N);
   end;

procedure addbruch (a, b: bruch; var e: bruch);
   begin
      e.Z:= a.Z * b.N + b.Z * a.N;
      e.N:= a.N * b.N;
   end;
```

Der Aufruf einer Prozedur erfolgt durch eine Prozeduranweisung:

> Prozedurname
> (Aktuelle Argumentliste) { kann entfallen }

Eine Prozeduranweisung bewirkt die Ausführung des Anweisungsteiles der aufgerufenen Prozedur nach Behandlung der Argumente in folgender Weise:

- Die aktuellen Argumente werden den formalen Argumenten in der physikalischen Reihenfolge zugeordnet. Bei Referenzaufruf werden dabei die Regeln der Typverträglichkeit, bei Wertaufruf die Regeln der Zuweisungsverträglichkeit angewendet (vgl. Abschnitt 2.3.5).
- Die formalen Argumente für Variablen mit Referenzaufruf dienen während der Prozedurausführung als Zugriff auf die entsprechenden aktuellen Variablen.
- Den formalen Argumenten für Variablen mit Wertaufruf wird Speicherplatz zugeordnet und sie erhalten vor Ausführung des Anweisungsteiles der Prozedur die Werte der entsprechenden aktuellen Argumente (Ausdrücke) zugewiesen.
- Die formalen Argumente für Prozeduren und Funktionen dienen während der Prozedurausführung als Namen für die entsprechenden aktuellen Prozeduren und Funktionen.

PASCAL–XSC

PASCAL–XSC erlaubt einen modifizierten Referenzaufruf in Verbindung mit strukturierten Datentypen (siehe Abschnitt 2.7.9).

Beispiel 2.7.2:

```
var
   a, b, e : bruch;
begin
   readbruch (a);
   readbruch (b);
   addbruch (a,b,e);
   ...
end.
```

Im Anweisungsteil einer Prozedur dürfen die Aufrufe von lokalen und beliebigen globalen Unterprogrammen, insbesondere der Prozedur selbst (rekursive Prozedur) auftreten. Dieser rekursive Aufruf kann sowohl direkt als auch indirekt erfolgen. Grundsätzlich muß eine gerufene Prozedur physikalisch vor der Stelle des Aufrufs vereinbart werden. Dies kann auch unvollständig in Form einer **forward**-Vereinbarung (siehe Abschnitt 2.7.8) erfolgen.

PASCAL–XSC

Anstelle der Konform-Array-Schemata steht das machtvollere Konzept der dynamischen Felder zur Verfügung (vgl. 2.3.2). Durch die Verwendung eines dynamischen Typs für ein Variablenargument bleiben die Indexbereiche variabel. Der Zugriff auf die Indexgrenzen erfolgt im Prozedurrumpf mittels der Funktionen *lbound* und *ubound*.

Beispiel 2.7.3:

```
type dynvektor = dynamic array [*] of real;
...
procedure  vekadd (var x, y, res: dynvektor);
  { Es werden gleiche Indexgrenzen von x, y und res vorausgesetzt }
  var i: integer;
  begin
    for i:= lbound(x) to ubound(x) do
      res[i] := x[i] + y[i]
  end;
```

Der Aufruf der Prozedur *vekadd* kann nun mit Vektoren vom dynamischen Typ *dynvektor* erfolgen. Dabei müssen bei dieser Implementierung die Indexbereiche der aktuellen Argumente *x*, *y* und *res* zusammen passen.

Tritt eine Funktion mit dynamischem Funktionsergebnis als formales Argument einer Prozedur auf, so darf der Funktionskopf in der Spezifikation nur den Namen des dynamischen Typs und keine Indexgrenzen enthalten.

2.7.2 Liste der Standardprozeduren und der Ein-/Ausgabeanweisungen

Im folgenden sind die im Sprachkern von PASCAL bzw. PASCAL–XSC vorhandenen Standardprozeduren und Ein-/Ausgabeanweisungen aufgeführt.

Erzeugung und Freigabe von referierten Variablen:

new (Pointervariable)
new (Pointervariable, Auswahlmarke, ..., Auswahlmarke)
dispose (Pointervariable)
dispose (Pointervariable, Auswahlmarke, ..., Auswahlmarke)

Lesen und Schreiben auf Filevariablen:

reset (Filevariable)
get (Filevariable)
read (Filevariable, Variable, ..., Variable)
readln (Textfilevariable, Variable, ..., Variable)
rewrite (Filevariable)
put (Filevariable)

write (Filevariable, Variable, ..., Variable)
writeln (Textfilevariable, Variable, ..., Variable)
page (Textfilevariable)

PASCAL–XSC

Erzeugung und Freigabe von referierten Variablen:

mark (Pointervariable)
release (Pointervariable)

Lesen und Schreiben auf Filevariablen:

reset (Filevariable, Stringausdruck)
rewrite (Filevariable, Stringausdruck)

Verändern der aktuellen Länge von String Variablen:

setlength (String Variable, Integer Ausdruck)

2.7.3 Funktionen

Für einen Teilalgorithmus, welcher immer nur ein Ergebnis von einfachem Typ (*integer, real, boolean, char*, Aufzählungstyp oder Pointertyp) liefert, kann anstelle einer Prozedur auch eine Funktion (Funktionsprozedur) formuliert werden:

```
function Name
   (Formale Argumentliste)   { kann entfallen }
   : Typ;
Vereinbarung; ···
begin
   Anweisung; ···
end;
```

Im Gegensatz zu Prozeduren wird bei einer Funktion ein Funktionsergebnis über den Namen der Funktion und nicht über ein formales Argument weitergegeben. Der Typ der Funktion (bzw. des Funktionsergebnisses) wird im Anschluß an die formale Argumentliste nach dem Doppelpunkt (:) spezifiziert. Der Funktionswert muß im Anweisungsteil der Funktion an den Funktionsnamen zugewiesen werden. Dazu darf der Funktionsname auf der linken Seite von Wertzuweisungen auftreten.

Man beachte, daß das Auftreten auf der rechten Seite einer Wertzuweisung den rekursiven Aufruf der Funktion bedeuten würde. Alle anderen Festlegungen für die Vereinbarung von Funktionen gelten ansonsten analog wie bei Prozeduren.

Der Aufruf einer Funktion erfolgt in der Form

> Funktionsname
> (Aktuelle Argumentliste) { kann entfallen }

als Operand in einem entsprechenden Ausdruck, wie dies von Standardfunktionen bereits bekannt ist. Die Auswertung des Ausdruckes wird unterbrochen, nach Behandlung der Argumente (wie bei Prozeduren auf Seite 90 beschrieben) wird der Anweisungsteil der Funktion ausgeführt und anschließend die Auswertung des Ausdruckes mit dem Funktionsergebnis anstelle des Funktionsaufrufes fortgesetzt.

PASCAL–XSC

PASCAL–XSC erlaubt einen modifizierten Referenzaufruf in Verbindung mit strukturierten Datentypen (siehe Abschnitt 2.7.9).

Im Anweisungsteil einer Funktion dürfen die Aufrufe von lokalen und beliebigen globalen Unterprogrammen, insbesondere der Funktion selbst (rekursive Funktion), auftreten. Dieser rekursive Aufruf kann sowohl direkt als auch indirekt über den Aufruf einer weiteren Funktion erfolgen. Grundsätzlich muß eine aufgerufene Funktion physikalisch vor der Stelle des Aufrufs vereinbart werden. Dies kann auch unvollständig in Form einer **forward**-Vereinbarung (siehe Abschnitt 2.7.8) erfolgen.

2.7.4 Funktionen mit allgemeinem Ergebnistyp

PASCAL–XSC

Funktionsergebnisse müssen nicht von einfachem Typ oder Pointertyp sein, sondern können von beliebig strukturiertem Typ sein. Die Wertzuweisung an das strukturierte Funktionsergebnis kann als Ganzes oder komponentenweise erfolgen, für einen Record-Typ auch innerhalb einer **with**-Anweisung.

Beispiel 2.7.4:

```
type komplex = record re, im : real end;
...
function kompladd (y,w: komplex) : komplex;
begin
   kompladd.re := y.re + w.re;
   kompladd.im := y.im + w.im
end;
```

PASCAL–XSC

Darüber hinaus kann für das Funktionsergebnis auch ein dynamischer Typ verwendet werden, bei dem alle Indexgrenzen durch Ausdrücke spezifiziert sind, welche beim Funktionsaufruf vor der Ausführung des Funktionsrumpfes auswertbar sein müssen.

Beispiel 2.7.5:

```
type dynvektor = dynamic array [*] of real;
...
function vekadd (x,y: dynvektor) :
            dynvektor [lbound(x)..ubound(x)]; {Funktionstyp}
{ Es werden gleiche Indexbereiche von x und y vorausgesetzt }
var i: integer;
begin
  for i:= lbound(x) to ubound(x) do
    vekadd[i] := x[i] + y[i]
end;
```

Tritt eine Funktion mit dynamischem Funktionsergebnis als formales Argument einer Prozedur auf, so darf der Funktionskopf in der Spezifikation nur den Namen des dynamischen Ergebnistyps und keine Indexgrenzen enthalten.

2.7.5 Liste der Standardfunktionen

Im folgenden sind die im Sprachkern von PASCAL bzw. PASCAL–XSC vorhandenen Standardfunktionen, gegliedert nach den zulässigen Typen für die Argumente, aufgeführt. Der jeweilige Ergebnistyp ist rechts in Kommentarklammern angegeben.

Argumenttyp *integer, boolean, char, code*

ord (Ausdruck)	{ *integer* }
succ (Ausdruck)	{ Argumenttyp }
pred (Ausdruck)	{ Argumenttyp }

Argumenttyp *integer*

odd (Ausdruck)	{ *boolean* }
chr (Ausdruck)	{ *char* }

Argumenttyp *integer, real*

abs (Ausdruck)	{ Argumenttyp }
sqr (Ausdruck)	{ Argumenttyp }
sqrt (Ausdruck)	{ *real* }

exp (Ausdruck)	{ *real* }
ln (Ausdruck)	{ *real* }
arctan (Ausdruck)	{ *real* }
sin (Ausdruck)	{ *real* }
cos (Ausdruck)	{ *real* }
round (Ausdruck)	{ *integer* }
trunc (Ausdruck)	{ *integer* }

Argumenttyp File

eof (Filevariable) bzw. eof	{ *boolean* }
eoln (Textfilevariable) bzw. eoln	{ *boolean* }

Argumenttyp beliebig

loc (Variable) { *integer* }

Argumenttyp Pointer

ord (Ausdruck) { *integer* }

Argumenttyp *integer, real, dotprecision*

sign (Ausdruck) { *integer* }

Argumenttyp *real* { Ergebnistyp *real* }

succ (R Ausdruck)
pred (R Ausdruck)
exp2 (R Ausdruck)
exp10 (R Ausdruck)
log2 (R Ausdruck)
log10 (R Ausdruck)
tan (R Ausdruck)
cot (R Ausdruck)
arcsin (R Ausdruck)
arccos (R Ausdruck)
arccot (R Ausdruck)
arctan2 (R Ausdruck, R Ausdruck)
sinh (R Ausdruck)
cosh (R Ausdruck)
tanh (R Ausdruck)
coth (R Ausdruck)
arsinh (R Ausdruck)
arcosh (R Ausdruck)
artanh (R Ausdruck)
arcoth (R Ausdruck)

PASCAL–XSC

Argumenttyp Array { Ergebnistyp: Indextyp des Arrays }

lbound (Array-Variable, Integer-Konstante)
lb (Array-Variable, Integer-Konstante)
lbound (Array-Variable)
lb (Array-Variable)
ubound (Array-Variable, Integer-Konstante)
ub (Array-Variable, Integer-Konstante)
ubound (Array-Variable)
ub (Array-Variable)

Argumenttyp *integer, real* { Ergebnistyp *string* }

image (I Ausdruck)
image (I Ausdruck, I Ausdruck)
image (R Ausdruck)
image (R Ausdruck, I Ausdruck)
image (R Ausdruck, I Ausdruck, I Ausdruck)
image (R Ausdruck, I Ausdruck, I Ausdruck, I Ausdruck)

Argumenttyp *string*

ival (ST Ausdruck)	{ *integer* }
ival (ST Ausdruck, ST Variable)	{ *integer* }
rval (ST Ausdruck)	{ *real* }
rval (ST Ausdruck, ST Variable)	{ *real* }
rval (ST Ausdruck, I Ausdruck)	{ *real* }
rval (ST Ausdruck, I Ausdruck, ST Variable)	{ *real* }
length (ST Ausdruck)	{ *integer* }
maxlength (ST Variable)	{ *integer* }
pos (ST Ausdruck, ST Ausdruck)	{ *integer* }
substring (ST Ausdruck, I Ausdruck, I Ausdruck)	{ *string* }

Zusätzliche Standardfunktionen für die Datentypen *complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix* und *cimatrix* werden vorübersetzt in den Modulen C_ARI, I_ARI, CI_ARI, MV_ARI, MVC_ARI, MVI_ARI und MVCI_ARI zur Verfügung gestellt (siehe dazu Kapitel 3).

2.7.6 Operatoren

PASCAL–XSC

Neben Prozeduren und Funktionen bietet PASCAL–XSC die Möglichkeit, Unterprogramme in Form von Operatoren zu definieren. Dabei werden grundsätzlich zwei Arten von Operatoren unterschieden, nämlich Operatoren *mit Ergebnis* und Operatoren *ohne Ergebnis*. Zur zweiten Art von Operatoren zählt lediglich der Zuweisungsoperator :=, dessen Definitions- bzw. Überladungsmöglichkeiten in Abschnitt 2.7.12 beschrieben werden. Auf den erstgenannten Typus von Operatoren soll in diesem Abschnitt eingegangen werden.

Ähnlich den Standardoperatoren können ein- oder zweistellige Operatoren für beliebige Operanden und mit beliebigem Ergebnistyp in allgemeinen Ausdrücken verwendet werden. Solche Operatoren müssen zunächst vereinbart werden in der Form:

operator Monadischer Operator (Formaler Operand)
            Resultatsname : Typspezifikation;
Vereinbarung; ...
begin
  Anweisung; ...
end;

bzw.

operator Dyadischer Operator (Formaler Operand, Formaler Operand)
            Resultatsname : Typspezifikation;
Vereinbarung; ...
begin
  Anweisung; ...
end;

Die Vereinbarung ähnelt ganz der Funktionsvereinbarung. Allerdings wird die Rolle des Funktionsnamens vom Resultatsnamen übernommen. Die Ergebniszuweisung muß im Operatorrumpf an diesen erfolgen. Die Angabe des formalen Operanden entspricht der des formalen Arguments bei Funktionen oder Prozeduren:

var { kann entfallen }
   Name : Typspezifikation

PASCAL–XSC

Haben beide Operanden denselben Typ und sollen beide mit Referenzaufruf oder Wertaufruf behandelt werden, so kann die Spezifikation durch

var { kann entfallen }
Name, Name : Typspezifikation

abgekürzt werden.

Als monadische Operatoren stehen

+, −, **not**	(Priorität 3)

zur Verfügung, als dyadische Operatoren

=, <>, <=, >=, <, >, **in**, ><	(Priorität 0)
+, +<, +>, −, −>, −<, +∗, **or**	(Priorität 1)
∗, ∗<, ∗>, /, /<, />, ∗∗, **mod**, **div**, **and**	(Priorität 2)

Darüber hinaus können benutzerdefinierte

Namen

als Operatorsymbole vereinbart werden. Neue Operatornamen müssen zuerst in einer Prioritätsdefinition auftreten:

priority Name = Prioritätszeichen; ...

Als Prioritätszeichen sind =, +, ∗ und ↑ zugelassen. Durch die Zeichen =, + und ∗ wird ein Name als dyadisches Operatorsymbol mit der Priorität 0 (=), 1 (+) und 2 (∗) festgelegt, während das Zeichen ↑ für monadische Operatorsymbole mit Priorität 3 steht.

Wird zu einem bereits definierten Operatorsymbol eine Vereinbarung mit neuen Operandentypen vorgenommen, so spricht man vom Überladen dieses Operators. So ist etwa der Operator + standardmäßig bereits überladen (einerseits Addition für *integer*, andererseits für *real*); er wird im nachfolgenden Beispiel zusätzlich mit der Vektoraddition überladen. Wird dagegen zu einem bereits definierten Operator eine Vereinbarung zu denselben Operandengtypen etwa in einem untergeordneten Block vorgenommen, so entspricht dies dem *Verdecken* des Operators in seiner ursprünglichen Bedeutung nach den Regeln der Blockstruktur (vgl. auch Abschnitt 2.7.10).

PASCAL–XSC

Beispiel 2.7.6:

```
type
   komplex = record re, im : real end;
   dynvektor = dynamic array [*] of real;
...
operator * (z, w: komplex) komplmult: komplex;
   begin
      komplmult.re := z.re * w.re - z.im * w.im;
      komplmult.im := z.re * w.im + z.im * w.re
   end;
...
operator + (x, y: dynvektor) vekadd: dynvektor [lb(x)..ub(x)];
   { Es werden gleiche Indexgrenzen von x und y vorausgesetzt }
   var i: integer;
   begin
      for i:= lbound(x) to ubound(x) do
         vekadd[i] := x[i] + y[i]
   end;
...
priority xor = +; {exklusives „oder“}
operator xor (a, b: boolean) exor: boolean;
   begin
      exor := a <> b
   end;
```

Der „Aufruf“ eines Operators erfolgt in der Form

Operatorsymbol Aktueller Operand

für einen monadischen Operator bzw. in der Form

Aktueller Operand Operatorsymbol Aktueller Operand

für einen dyadischen Operator. Operatoraufrufe können wie Funktionsaufrufe nur innerhalb von Ausdrücken auftreten. Die Auswertung des Ausdruckes wird unterbrochen, nach Behandlung der aktuellen Operanden (wie bei Prozeduren auf Seite 90 beschrieben) wird der Anweisungsteil des Operators ausgeführt und anschließend die Auswertung des Ausdruckes mit dem Operatorergebnis anstelle des Operatoraufrufes fortgesetzt.

PASCAL–XSC erlaubt einen modifizierten Referenzaufruf in Verbindung mit strukturierten Datentypen (siehe Abschnitt 2.7.9).

PASCAL–XSC

Beispiel 2.7.7:

```
...
var
  k1, k2, k3, k4: komplex;
  vekx, veky, vekz: dynvektor [1..100];
  b1, b2, b3: boolean;
begin
  ...
  k4 := k1 * k2 * k3;
  ...
  vekz := vekx + veky;
  ...
  b3 := b1 xor b2;
  ...
end.
```

Im Anweisungsteil eines Operators dürfen wie bei Funktionen und Prozeduren die Aufrufe von lokalen und beliebigen globalen Unterprogrammen, insbesondere des Operators selbst (rekursiver Operator) auftreten. Dieser rekursive Aufruf kann sowohl direkt als auch indirekt erfolgen. Grundsätzlich muß ein verwendeter Operator physikalisch vor der Stelle des Aufrufs vereinbart werden. Dies kann auch unvollständig in Form einer **forward**-Vereinbarung (siehe Abschnitt 2.7.8) erfolgen.

2.7.7 Tabelle der Standardoperatoren

PASCAL–XSC

Im folgenden sind die im Sprachkern von PASCAL–XSC vorhandenen Standardoperatoren (PASCAL-Standardoperatoren einschließlich Erweiterungen von PASCAL–XSC) in einer Verknüpfungstabelle zusammengefaßt. Die darüber hinaus in den Arithmetikmodulen zusätzlich bereitgestellten Operatoren für die arithmetischen Standardtypen *complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix* und *cimatrix* sind darin nicht erfaßt. Entsprechende Tabellen finden sich in Kapitel 3 und in den Verzeichnissen im Anhang B.4.

PASCAL–XSC

linker Operand \ rechter Operand	integer	real	boolean	char	string	**set**
monadisch	$+, -$	$+, -$	**not**			
integer	$\circ, \circ<, \circ>$, **div, mod**, $\vee$	$\circ, \circ<, \circ>$, $\vee$				**in**
real	$\circ, \circ<, \circ>$, $\vee$	$\circ, \circ<, \circ>$, $\vee$				
boolean			**or, and**, $=, <>$, $<=, >=$			**in**
char				$+$ $\vee$	$+$ $\vee$ **in**	**in**
string				$+$ $\vee$	$+$ $\vee$ **in**	
set						$+, -, *$, $=, <>$, $<=, >=$
Code						**in**

Die Standardoperatoren von PASCAL–XSC
$\circ \in \{+, -, *, /\}$
$\vee \in \{=, <>, <, <=, >, >=\}$

2.7.8 forward- und external-Vereinbarung

PASCAL–XSC

Die sogenannte **forward**-Vereinbarung ermöglicht den gegenseitigen und rekursiven Aufruf von Unterprogrammen. Mit ihr kann die Vereinbarung von Prozeduren, Funktionen und Operatoren auch unvollständig in Form des Prozedur- bzw. Funktions- bzw. Operatorkopfes mit nachfolgendem **forward** anstelle des Prozedur- bzw. Funktions- bzw. Operatorrumpfes erfolgen.

PASCAL–XSC

Für ein solches **forward**-deklariertes Unterprogramm muß die vollständige Vereinbarung im selben Vereinbarungsteil erfolgen. Diese wird wiederum durch die Wortsymbole **procedure** bzw. **function** bzw. **operator** und den jeweiligen Prozedur- bzw. Funktions- bzw. Operatornamen eingeleitet. Die formale Argumentliste, der Resultatsname (bei Operatoren) und die Ergebnistypangabe (bei Funktionen und Operatoren) müssen, im Gegensatz zu Standard-PASCAL, ebenfalls wieder angegeben werden.

Durch eine **external**-Vereinbarung können getrennt übersetzte Prozeduren, Funktionen und Operatoren, die in einer anderen Programmiersprache oder in Maschinensprache geschrieben sind, eingebunden werden. Wie bei der **forward**-Vereinbarung steht anstelle von Vereinbarungsteil und Rumpf des Unterprogramms das Wortsymbol **external** optional gefolgt von einer Stringkonstanten. Der Name des externen Unterprogramms ist dabei entweder der Prozedur- bzw. der Funktions- bzw. der Resultatsname (bei Operatoren) des entsprechenden Unterprogramms oder aber der Wert der hinter dem Wortsymbol **external** stehenden Stringkonstanten. Somit können externe Unterprogramme auch überladen werden. Die Angabe der formalen Argumentliste dient lediglich dem Compiler zur Kontrolle der Aufrufe bei der Verwendung der Routinen. Näheres zu der Verwendung von externen Unterprogrammen in Verbindung mit **external** muß dem Benutzerhandbuch zur jeweiligen Implementierung entnommen werden.

2.7.9 Modifizierter Referenzaufruf für strukturierte Datentypen

PASCAL–XSC

Üblicherweise werden Operatoren oder auch Funktionen geschachtelt angewendet, d. h. innerhalb eines Ausdrucks werden mehrfach Operatoren oder Funktionen aufgerufen. Somit müssen als aktuelle Operanden bzw. aktuelle Argumente Ausdrücke zugelassen sein. Im strengen PASCAL-Sinne bedeutet dies, daß die formalen Operanden bzw. die formalen Argumente als Werteparameter vereinbart und entsprechend behandelt werden müssen. Insbesondere wird dann beim Aufruf eines so vereinbarten Operators bzw. einer Funktion für jedes Werteargument lokaler Speicher angelegt, was bei umfangreichen strukturierten Datentypen ineffizient ist.

Um dies zu vermeiden, ist es in PASCAL–XSC für die strukturierten Datentypen möglich, beim Aufruf von Prozeduren, Funktionen und Operatoren mit formalen Referenzargumenten auch Ausdrücke als aktuelle Argumente zu verwenden. Das formale Argument für den Referenzaufruf dient in diesem Fall während der Unterprogrammausführung als Zugriff auf die anonyme Hilfsgröße, die vom Compiler bereits bei der Ausdrucksauswertung angelegt wurde und den Wert des Ausdrucks enthält.

PASCAL–XSC

Beispiel 2.7.8:

Mit den Vereinbarungen

```
const n = 100;
type matrix = array [1..n, 1..n] of real;
var m1, m2, m3, m4, m5 : matrix;
operator + (var a, b: matrix) resplus : matrix;
  var i, j: integer;
  begin
    for i:= 1 to n do
      for j:= 1 to n do
        resplus[i,j] := a[i,j] + b[i,j];
  end;
function comp_sqr (var a: matrix) : matrix;
  var i, j: integer;
  begin
    for i:= 1 to n do
      for j:= 1 to n do
        comp_sqr[i,j] := sqr (a[i,j]);
  end;
```

ist auch eine Anweisung der Form

```
m1 := m1 + m2 + comp_sqr (m3 + m4 + m5);
```

möglich.

2.7.10 Überladen von Prozeduren, Funktionen und Operatoren

PASCAL–XSC

In PASCAL–XSC werden Prozeduren, Funktionen und Operatoren an ihrem Namen (bzw. am Operatorzeichen) und an Anzahl, Typ und Reihenfolge ihrer Parameter erkannt. Es können somit im Gegensatz zu Standard-PASCAL mehrere Prozeduren, Funktionen und Operatoren mit demselben Namen innerhalb eines Blocks definiert werden. Diese müssen lediglich anhand ihrer Argumente unterschieden werden können. Man bezeichnet dies als *Überladen* der Namen.

PASCAL–XSC

Wollte man in Standard-PASCAL eine Sinus-Funktion für komplexe Zahlen implementieren, so müßte man diese etwa *csin* nennen. In PASCAL–XSC hingegen kann der Standardfunktionsname *sin* überladen und auch z. B. für Argumente vom selbstdefinierten Typ *komplex* aus dem Beispiel 2.7.6 verwendet werden.

Durch diese Überladungsmöglichkeit wird ein generisches Namenskonzept in die Sprache eingeführt, das es dem Benutzer erlaubt, die vordefinierten Bezeichner wie z. B. *sin*, *cos*, *exp* oder *ln* auch für beliebige Datentypen zu verwenden.

Beispiel 2.7.9:

```
type komplex = record re, im : real end;
...
function exp (c : komplex) : komplex;
begin
   exp.re:= exp (c.re) * cos (c.im);
   exp.im:= exp (c.re) * sin (c.im);
end;
```

Zu beachten ist in diesem Beispiel, daß im Rumpf der Funktion die *reelle* Standardfunktion *exp* mit dem reellen Argument *c.re* aufgerufen wird. Dies ist somit kein rekursiver Aufruf der neu definierten Funktion *exp*.

Für das Überladen von Prozeduren, Funktionen und Operatoren gelten die folgenden wichtigen Regeln:

- Die formalen Argumentlisten von überladenen Unterprogrammen müssen sich unterscheiden, d. h. die Argumente dürfen nicht in Anzahl, Typ und Reihenfolge zugleich übereinstimmen. Dabei ist es unerheblich, ob es sich um Werte- oder Variablenparameter handelt (verträgliche Typen gelten als gleich).
- Der Ergebnistyp von Funktionen und Operatoren wird nicht für die Identifikation herangezogen.
- Funktionen dürfen nur mit Funktionen, Operatoren nur mit Operatoren, Prozeduren nur mit Prozeduren überladen werden.
- Ein Unterprogrammname darf innerhalb desselben Blocks nicht mehr als Konstanten-, Variablen- oder Typname verwendet werden.

Die Regelung bezüglich des Verdeckens von Namen gilt in PASCAL–XSC entsprechend wie in Standard-PASCAL.

PASCAL–XSC

Ein Name wird also verdeckt, wenn in einem inneren Block ein Objekt gleichen Namens vereinbart wird. Unterprogramme des äußeren Blocks werden jedoch nicht verdeckt, wenn sie im inneren Block lediglich mit anderen Argumentlisten überladen werden.

Beim Aufruf eines überladenen Unterprogramms gilt es, die folgenden Regeln zu beachten:

- Beim Referenzaufruf müssen die aktuellen Argumente stets mit den formalen Argumenten verträglich sein.
- Beim Wertaufruf brauchen die aktuellen Argumente lediglich zuweisungsverträglich zu den formalen Parametern sein. Falls kein Unterprogramm mit typverträglichen Argumenten vorhanden ist, können die zuweisungsverträglichen aktuellen Argumente automatisch konvertiert werden. Hierbei ist zuweisungsverträglich im strengen Sinne zu verstehen, d. h. durch überladene Zuweisungen „zuweisungsverträglich gemachte“ Typen gelten weiterhin als *nicht zuweisungsverträglich beim Wertaufruf*, sie können also nicht automatisch konvertiert werden (vgl. Abschnitt 2.3.5 und Abschnitt 2.7.12).

Kommen bei einem Aufruf mehrere überladene Prozeduren bzw. Funktionen bzw. Operatoren in Frage, so werden zuerst die Variablenparameter und die nicht konvertierbaren Werteparameter zugeordnet. Danach wird versucht, die restlichen Werteparameter ohne Konversion zuzuordnen. Falls dies nicht möglich ist, wird von zwei Unterprogrammen, deren formale Parameterlisten zu der aktuellen passen, dasjenige ausgewählt, welches den ersten *besser passenden* Parameter besitzt. Besser passend heißt dabei, daß die Typen nicht nur anpaßbar, sondern verträglich sind.

Beispiel 2.7.10:

```
operator +* (a: integer; b: real) ir_res: real;
...
operator +* (a: real; b: integer) ri_res: real;
...
var
  i : integer;
  r, erg : real;
...
erg:= i +* r;   { 1. Operator wird verwendet }
erg:= r +* i;   { 2. Operator wird verwendet }
erg:= i +* i;   { 1. Operator wird verwendet }
erg:= r +* r;   { Anweisung nicht möglich }
```

2.7.11 Überladen von *read* und *write*

PASCAL–XSC

Das im letzten Abschnitt beschriebene Überladungsprinzip gilt natürlich auch für die Standardprozeduren *read* und *write*. Aufgrund der Tatsache, daß diese Prozeduren aber bereits in Standard-PASCAL einige spezielle Eigenschaften haben, wurde das Konzept des Überladens für diese Ein-/Ausgaberoutinen entsprechend modifiziert.

Wie bereits in Abschnitt 2.5.2 beschrieben, erlauben die Standardprozeduren *read* und *write* in Verbindung mit Textfiles im allgemeinen

- einen optionalen ersten Parameter vom Typ *text*
- eine beliebige Anzahl von unterschiedlichen Parametern
- zusätzliche, optionale Formatspezifikationen durch Doppelpunkt getrennt hinter einem Ein-/Ausgabeelement

PASCAL–XSC erlaubt es nun, diese Eigenschaften auch auf benutzerdefinierte Ein- und Ausgabeprozeduren durch Überladung von *read* und *write* zu übertragen. Bei der Deklaration und beim Aufruf sind gewisse Regeln zu beachten.

Deklaration

Bei der Vereinbarung einer neuen Prozedur für die Ein-/Ausgabe muß der erste formale Parameter stets ein **var**-Parameter vom Typ *text* oder von einem beliebigen Filetyp sein. Der zweite Parameter stellt das ein- bzw. auszugebende Objekt dar und darf kein File sein. Alle eventuell folgenden Parameter sind als Formatspezifikationen zum zweiten Parameter zu verstehen.

Beispiel 2.7.11:

> **type** interval = **record** inf, sup : real **end**;
> ...
> **procedure** write (**var** f: text; int: interval; m,n: integer);

Aufruf

Beim Aufruf der so vereinbarten Prozedur kann der Fileparameter wie bei den Standardprozeduren entfallen, was einem Aufruf mit den Standardfiles *input* bzw. *output* entspricht. Wird ein Fileparameter angegeben, so ist nach einem Komma als zweiter aktueller Parameter das Ein-/Ausgabeargument anzugeben. Jeweils mit Doppelpunkt als Trennzeichen folgen dann die Formatparameter.

PASCAL–XSC

Beispiel 2.7.12:

Aufrufvarianten für *int* vom Typ *interval* und *f* vom Typ *text* wären mit obiger Vereinbarung z. B.

write (int : 10 : 5); oder write (f, int : 12 : 6);

Mehrere solche Anweisungen können entsprechend Standard-PASCAL zu einer Anweisung mit beliebigen Ein-/Ausgabeargumenten, für die *read*-/*write*-Prozeduren definiert sind, zusammengefaßt werden.

Beispiel 2.7.13:

Mit einer *real*-Variablen *a* wäre die Anweisung

writeln (f, a : 20 : 9, int : 50 : 10, true : 4);

äquivalent zur Anweisungsfolge

```
write (f, a : 20 : 9);
write (f, int : 50 : 10);
write (f, true : 4);
writeln (f);
```

Der Compiler sucht für jeden dieser *write*-Aufrufe nach einer benutzerdefinierten Prozedur mit passender Parameterzahl und passenden Parametertypen, wobei jeder Doppelpunkt als Komma interpretiert wird. Findet er keine solche Prozedur, so wird, falls möglich, die Standard-Ein-/Ausgabeprozedur verwendet.

Um die Eingabe oder die Ausgabe mit unterschiedlicher Anzahl von Formatparametern zuzulassen, muß der Benutzer für jede Anzahl von Parametern eine eigene Prozedur implementieren.

Beispiel 2.7.14:

Will man die Rundung bei der Ausgabe von *real*-Zahlen nicht über den standardmäßigen *integer*-Parameter steuern, so könnte man folgende Prozeduren implementieren:

```
procedure write (var f: text; r: real;
                 w, n: integer; rd: boolean);
  begin
    if rd then
      write (f, r : w : n : +1)
    else
      write (f, r : w : n : −1);
  end;
```

PASCAL–XSC

Beispiel 2.7.15:

Weitere Varianten für die Formatangaben:

```
procedure write (var f: text; r: real;
                  w: integer; rd: boolean);
  begin
    write (f, r : w : 0 : rd)
  end;
procedure write (var f: text; r: real; rd: boolean);
  begin
    write (f, r : 20 : 0 : rd)
  end;
```

Damit könnten z. B. die *real*-Ausdrücke *a, b, c* ausgegeben werden durch

```
writeln (output, a : 10 : 5 : true, b : 10 : false, c : true);
```

Ein abschließendes Beispiel soll nochmals demonstrieren, daß mit diesem Überladungskonzept für *read* und *write* dem Benutzer kaum Grenzen gesetzt sind.

Beispiel 2.7.16:

```
const
  format1 = '[ ]';
  format2 = '<>';
  format3 = '( )';
...
procedure write (var f: text; int: interval; klammerung: string);
var l, r: char;
begin
  l:= klammerung[1];
  r:= klammerung[2];
  write (f, l, int.inf : 20 : 13, ',', int.sup : 20 : 13, r);
end;
```

Mit diesen Vereinbarungen lassen sich Intervalle in unterschiedlichen Formen ausgeben:

mit	write (int : format1);	in der Form	[... , ...]
mit	write (int : format2);	in der Form	< ... , ... >
mit	write (int : format3);	in der Form	(... , ...)

Mit diesen Möglichkeiten zur Formatsteuerung könnte man sogar FORTRAN-ähnliche Formatangaben realisieren.

2.7.12 Überladen des Zuweisungsoperators :=

PASCAL–XSC

Eine wesentliche Vereinfachung gerade im Hinblick auf die Verwendung einer mathematischen Notation für Algorithmen bzw. Programme ergibt sich aus der Möglichkeit, den Zuweisungsoperator := in Form eines Operators ohne Ergebnis zu überladen. Damit kann die Wertzuweisung auch für zunächst nicht zuweisungsverträgliche Typen möglich gemacht werden.

Die Vereinbarung erfolgt in der Form

> **operator** := (Formaler Operand1, Formaler Operand2);
> Vereinbarung; ...
> **begin**
>   Anweisung; ...
> **end**;

und ähnelt der Prozedurvereinbarung. Von der in Abschnitt 2.7.6 beschriebenen Operatorvereinbarung unterscheidet sie sich dahingehend, daß die Angabe des Resultatsnamens und der Typspezifikation entfallen und daß der formale Operand 1 gemäß

> **var** Name : Typspezifikation

beschrieben sein *muß*, während der formale Operand 2 wie gewohnt durch

> **var** { kann entfallen }
>   Name : Typspezifikation

spezifiziert wird.

Im Anweisungsteil des Operators kann durch entsprechende Anweisungen festgelegt werden, wie die rechte Seite der Zuweisung (Operand 2) auf die linke Seite der Zuweisung (Operand 1) abgebildet wird. Der **var**-Parameter Operand 1 ist somit in der Regel Ausgabeparameter dieses Unterprogramms.

Der Aufruf eines überladenen Zuweisungsoperators erfolgt entsprechend der üblichen Wertzuweisung gemäß

> Variable := Ausdruck

wobei nun die linke und rechte Seite der Zuweisung entsprechend der durch die Überladung definierten Typkombination als zuweisungsverträglich anzusehen sind (vgl. Abschnitt 2.3.5).

PASCAL–XSC

Zu beachten ist, daß diese Zuweisungsverträglichkeit sich nicht auf den Wertaufruf von Unterprogrammen ausdehnt (vgl. Abschnitt 2.7.10, Seite 106), sondern nur für die eigentliche Zuweisung gilt.

Im nachfolgenden Beispiel wird anhand von Intervallen bzw. Vektoren demonstriert, wie durch die Überladung der Zuweisung das Arbeiten mit Werten aus dem im mathematischen Sinne eingebetteten Raum der reellen Zahlen (also Punktintervallen) oder auch die Initialisierung von Vektoren oder Matrizen erleichtert wird.

Beispiel 2.7.17:

```
...
var
   x  : interval;
   iv : ivector[1..n];
   im : imatrix[1..n,1..n];
...
operator := (var x: interval; r: real);    { Op1 }
   begin
      x.inf := r;
      x.sup := r;
   end;
operator := (var iv: ivector; r: real);    { Op2 }
   var i: integer;
   begin
      for i:= lb (iv) to ub (iv) do
         iv[i] := r;
   end;
operator := (var im: imatrix; r: real);    { Op3 }
   var i : integer;
   begin
      for i:= lb (im) to ub (im) do
         im[i]:= r; { Aufruf von Op2 }
   end;
...
x  :=  5.5;  { Op1-Aufruf, liefert Punktintervall }
iv :=  0;    { Op2-Aufruf, liefert Intervall-Nullvektor }
im :=  0;    { Op3-Aufruf, liefert Intervall-Nullmatrix }
```

2.8 Module

PASCAL–XSC

In Standard-PASCAL kann ein Programm nur aus einem einzigen Programmtext bestehen, der vollständig vorliegen muß, bevor er übersetzt und ausgeführt werden kann. PASCAL–XSC erlaubt es hingegen, ein Programm in mehrere Teile – Module genannt – zu zerlegen, die getrennt entwickelt und übersetzt werden können.

Module stellen vorrangig Sammlungen von Prozeduren, Funktionen, Operatoren und zugehörigen Konstanten- und Typdefinitionen sowie Variablenvereinbarungen dar. Die Vereinbarung von Modulen erfolgt ähnlich wie die von Programmen, ihre Compilierung erfolgt separat. Ein Modul hat folgende Syntax:

> **module** Name;
>   Use-Klausel; ...
>   **global** Vereinbarung; ... { **global** kann fehlen }
> **begin** { kann zusammen mit Anweisungsteil entfallen }
>   Anweisung; ...
> **end**.

Nach dem Wortsymbol **module** wird der Name des Moduls festgelegt.

Vereinbarungen haben dieselbe Form wie im Vereinbarungsteil eines Programmes. Wird eine Vereinbarung durch das Wortsymbol **global** eingeleitet, so sind alle darin vereinbarten Objekte globale Größen des Moduls, d. h. sie stehen zum Export in andere Module oder in das Hauptprogramm zur Verfügung. Alle anderen vereinbarten Objekte sind lokale Größen des Moduls.

In der Definition eines globalen Typs darf auf der rechten Seite des Gleichheitszeichens das Wortsymbol **global** auftreten. In diesem Fall steht auch die Struktur des globalen Typs zum Export zur Verfügung; z. B. werden durch die Typdefinition

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

sowohl der Typname *komplex* als auch die Recordstruktur und damit die Komponentennamen *re, im* global, während durch

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

die Struktur des Typs *komplex* lokal definiert wird und nur der Typ *komplex* exportiert wird. Zugriffe auf die Komponenten der Datenstruktur sind damit nur in dem Modul möglich, das die Typdefinition selbst enthält.

PASCAL–XSC

Beispiel 2.8.1:

Ein einfaches Modul zur Bereitstellung einer komplexen Arithmetik kann etwa folgende Form haben:

```
module KomplexeArithmetik;
   global type komplex = global record re, im: real end;
   global operator + (z, w: komplex) res: komplex;
      begin
         res.re := z.re + w.re;
         res.im := z.im + w.im
      end;
   global operator * (z, w: komplex) res: komplex;
      begin
         res.re := z.re * w.re − z.im * w.im;
         res.im := z.re * w.im + z.im * w.re
      end;
end.
```

Durch eine oder mehrere **use**-Klauseln können die globalen Größen der aufgezählten Module in einem Programm bzw. einem Modul bekannt gemacht werden (Import von Objekten). Tritt in einer Klausel das Wortsymbol **global** auf, so stehen alle durch diese Klausel importierten Objekte auch für den Export zur Verfügung. Eine **use**-Klausel hat die Form:

use global Modulnamensliste { **global** kann fehlen }

Beispiel 2.8.2:

Das folgende Modul stellt auf der Grundlage des Moduls *KomplexeArithmetik* eine Addition für komplexe Vektoren zur Verfügung:

```
module KomplexeVektorarithmetik;
   use global KomplexeArithmetik;
   global type
      komplexvektor = global dynamic array [*] of komplex;
   global operator + (x, y: komplexvektor)
                        res: komplexvektor [lbound(x)..ubound(x)];
      var i: integer;
      begin
         for i:= lbound(x) to ubound(x) do
            res[i] := x[i] + y[i]
      end;
end.
```

PASCAL–XSC

Man beachte, daß durch die use-Klausel

use KomlexeVektorarithmetik;

in einem anderen Modul oder im Hauptprogramm neben dem Typ *komplexvektor* und dem zugehörigen Operator + auch der Typ *komplex* und die zugehörigen Operatoren + und $*$ zur Verfügung stehen, da das definierende Modul durch

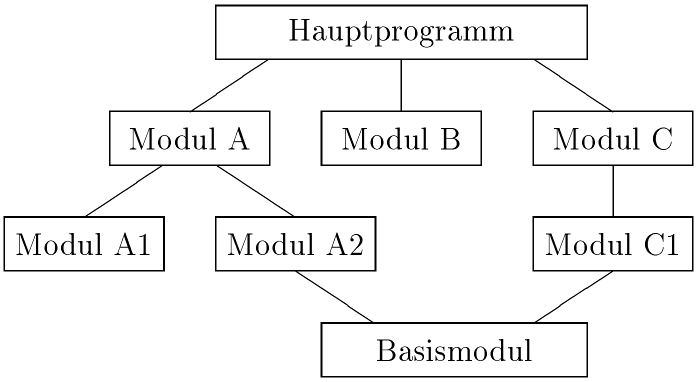
use global KomplexeArithmetik;

global eingezogen wurde. Würde dagegen im Modul *KomplexeVektorarithmetik* nur die Klausel

use KomplexeArithmetik;

verwendet, wäre diese Klausel auch in dem Modul oder Hauptprogramm notwendig, das *KomplexeVektorarithmetik* importiert.

Die **use**-Klauseln errichten eine sogenannte Modulhierarchie zwischen den einzelnen Modulen und dem Hauptprogramm. Die graphische Darstellung ist in einem azyklischen Graphen möglich. Dieser ähnelt einer Baumstrukur, an deren Wurzel das Hauptprogramm steht mit den Modulen, die ins Hauptprogramm importiert werden, als Söhne.



In Abweichung von der üblichen Baumstruktur dürfen verschiedene Module auch dasselbe Modul (in obiger Graphik z. B. das Basismodul) importieren. Ein gegenseitiger oder zyklischer Import ist jedoch, auch auf Umwegen, nicht erlaubt. Die Modulhierarchie legt die Reihenfolge der Übersetzung der Module fest. Ein Modul kann erst dann compiliert werden, wenn alle importierten Module zuvor übersetzt wurden. Dabei müssen in jedem zu übersetzenden Modul mindestens die exportierten Größen vereinbart bzw. definiert sein, während deren Implementierung noch nicht vollständig zu sein braucht. Bei Prozeduren sind z. B. leere Anweisungsteile ausreichend.

PASCAL–XSC

Die vollständige Implementierung der Module kann also nach der ersten Entwurfsphase, in welcher der Aufbau der einzelnen Module festgelegt und ihr Zusammenspiel überprüft wird, parallel durch möglicherweise verschiedene Teams ausgeführt werden.

Werden dabei am Definitionsteil eines Moduls, also an den Vereinbarungen der zum Export zur Verfügung stehenden Objekte, Veränderungen nicht mehr vorgenommen, genügt die Übersetzung dieses Moduls. Andernfalls müssen alle von diesem abhängigen Module, d. h. die in der Modulhierarchie darüber stehenden Module, neu übersetzt werden.

Ein Modul kann nach dem Vereinbarungsteil auch einen Anweisungsteil enthalten. Die Ausführung dieses Anweisungsteiles erfolgt genau einmal zu Beginn der Programmausführung. Dabei werden alle Module entsprechend der Modulhierarchie des Programmes berücksichtigt. Im Anweisungsteil eines Moduls können die lokalen und globalen Variablen des Moduls initialisiert werden. Hierzu kann ein beliebiges Programmstück mit den Größen des Moduls verwendet werden. In der oben angegebenen Modulhierarchie würde der Anweisungsteil des Basismoduls vor dem Anweisungsteil des Moduls A2 und dieser wiederum vor dem Anweisungsteil von Modul A ausgeführt werden.

2.9 Textverarbeitung

In Verbindung mit Textdateien vom Typ *text* sowie mit Zeichen (*char*) und Zeichenketten (**packed array** *[1..n]* **of** *char*) bietet Standard-PASCAL nur sehr eingeschränkte Möglichkeiten für Textverarbeitung. Außer den lexikalischen Vergleichen von Zeichenketten bzw. Zeichen gibt es keine Möglichkeit, Ausdrücke mit den genannten Datentypen zu formulieren. Während die Ausgabe sowohl für Zeichen als auch für Zeichenketten möglich ist, können nur *char*-Variablen, aber keine Zeichenkettenvariablen eingelesen werden.

PASCAL–XSC

Der dynamische Stringtyp (Abschnitt 2.3.2), der Stringausdruck (Abschnitt 2.4.3.2) und die Stringstandardfunktionen, zusammen mit Vergleichen, Wertzuweisung und Ein-/Ausgabe von Strings erlauben eine bequeme Textverarbeitung.

Für die bei der Textverarbeitung üblicherweise anfallenden Operationen und Zugriffe auf charakteristische Stringdaten stehen folgende Standardfunktionen und Standardprozeduren zur Verfügung:

function image (i: integer) : string;

> Liefert die dem Wert *i* entsprechende Ziffernfolge als String mit einer der Standardausgabe von *integer*-Werten entsprechenden Zeichenanzahl.

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

> Liefert die dem Wert *i* entsprechende Ziffernfolge als String mit mindestens *len* Zeichen (eventuell mit führenden Leerzeichen aufgefüllt), entsprechend der Ausgabe von *integer*-Werten.

function image (r: real) : string;

> Liefert die dem Wert *r* entsprechende Ziffernfolge als String mit einer der Standardausgabe von *real*-Werten entsprechenden Zeichenanzahl.

function image (r: real; width: integer) : string;

> Liefert die dem Wert *r* entsprechende Ziffernfolge als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) entsprechend der Ausgabe von *real*-Werten.

PASCAL–XSC

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

Liefert die dem Wert *r* entsprechende Ziffernfolge als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen aufgefüllt) und *fracs* Nachkommastellen, entsprechend der Ausgabe von *real*-Werten.

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

Liefert die dem Wert *r* entsprechende Ziffernfolge als String mit mindestens *width* Zeichen insgesamt, *fracs* Nachkommastellen und entsprechend *round* gerundet:

$$round \left\{ \begin{array}{ll} < 0 & \text{nach unten gerundet} \\ = 0 & \text{nächstliegend gerundet} \\ > 0 & \text{nach oben gerundet} \end{array} \right\}$$

function substring (s: string; p, l: integer) : string;

Liefert den Teilstring der Länge *l* aus *s* ab dem *p*-ten Zeichen.

function length (s: string) : integer;

Liefert die aktuelle Länge von *s*.

function maxlength (**var** s: string) : integer;

Liefert die maximale Länge von *s*.

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

Liefert die Position des ersten Auftretens von *sub* in *s* (0 bei Nichtauftreten).

function ival (s: string) : integer;

Liest *integer*-Wert aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *integer*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird ebenfalls vernachlässigt.

function ival (s: string; **var** rest: string) : integer;

Liest *integer*-Wert aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *integer*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird in *rest* zurückgegeben.

PASCAL–XSC

function rval (s: string) : real;

Liest *real*-Wert aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *real*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird ebenfalls vernachlässigt.

function rval (s: string; **var** rest: string) : real;

Liest *real*-Wert aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *real*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird in *rest* zurückgegeben.

function rval (s: string; round: integer) : real;

Liest *real*-Wert, gemäß *round* gerundet (vgl. *image*), aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *real*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird ebenfalls vernachlässigt.

function rval (s: string; round: integer; **var** rest: string) : real;

Liest *real*-Wert, gemäß *round* gerundet (vgl. *image*), aus *s* aus. Der String *s* enthält eine Zeichenfolge, die eine *real*-Konstante darstellt. Dabei werden führende Leerzeichen überlesen. Ein eventuell verbleibender Reststring wird in *rest* zurückgegeben.

procedure setlength (**var** s: string; lng: integer);

Setzt die aktuelle Länge der Stringvariablen *s* auf *lng*. Der Wert von *lng* muß im Bereich 0..*maxlength(s)* liegen.

Beispiel 2.9.1:

image (4728,5)	liefert	' 4728'
image (3.14159,7,4,1)	liefert	' 3.1416'
substring ('AAABB',3,3)	liefert	'ABB'
length ('abcde')	liefert	5
pos ('AB','AAABB')	liefert	3
ival ('512')	liefert	512
rval ('−1.5E6')	liefert	−1.5E+06

PASCAL–XSC

Als Vergleichsoperatoren stehen

=, <>, <=, <, >=, >

mit der üblichen Bedeutung der lexikographischen Anordnung zur Verfügung, wobei der kürzere, n Zeichen lange String stets vor (<) dem längeren angeordnet ist, wenn beide auf den ersten n Positionen übereinstimmen. Zusätzlich steht ein Vergleichsoperator

in

mit zwei String-Operanden zur Verfügung, wobei *s1* **in** *s2* den Wert *true* liefert, falls der String *s1* als Teilstring in *s2* auftritt, andernfalls ergibt sich der Wert *false*.

Beispiel 2.9.2:

Sei	s5 := 'AAABB';	
dann liefert	'A' **in** s5	den Wert *true*
und	'BBA' **in** s5	den Wert *false*.

Die Wertzuweisung an Stringvariablen

Stringvariable := Stringausdruck

ist stets möglich, wenn die Stringvariable vom Standardtyp *string* ist und der rechts stehende Ausdruck ein beliebiger Stringausdruck vom Typ *string* oder von einem Array-Zeichenkettentyp ist.

Ist die aktuelle Länge des rechtsstehenden Ausdruckes größer als die maximale Länge der linksstehenden Variablen, so werden die überzähligen Zeichen am Ende weggelassen.

Beispiel 2.9.3:

Es seien folgende Typen und Variablen gegeben:

type	string_10	=	string[10];
	string_20	=	string[20];
var	s5	:	string[5];
	s10	:	string_10;
	s20	:	string_20;
	s	:	string; {Länge implementierungsabhängig}

PASCAL–XSC

Beispiele für Wertzuweisungen wären dann:

```
s5   :=  'ABCDE'
s10  :=  s5;
s20  :=  'AABBCC';
s5   :=  s20; { s erhält den Wert 'AABBC' }
s20  :=  '';
s5   :=  'AAA' + 'BBB' { s5 erhält den Wert 'AAABB' }
```

2.9.1 Eingabe von Zeichen und Zeichenketten

Während sich PASCAL bei der Ausgabe von Zeichen oder Zeichenketten gemäß der Intuition des Programmierers verhält, weist das Einlesen dieser Datentypen von der Konsole oft unerwartete Eigenschaften auf.

Beim Lesen einer *char*-Variablen *c* mit der Anweisung

```
read (c);
```

wird, entsprechend der Definition von *read* (vgl. Abschnitt 2.5.2), die Sequenz

```
c := input↑;
get (input);
```

ausgeführt. Das bedeutet, daß der Pufferinhalt zunächst in die Variable *c* gespeichert und danach das nächste Zeichen in den Puffer gelesen wird. Mit dem ersten *read* auf *input* wird nun in die Variable *c* ein Leerzeichen gelesen, das dem („nullten“) Zeilenendezeichen entspricht. Der Grund dafür liegt darin, daß unmittelbar beim Programmstart *eoln (input) = true* gilt.

Beispiel 2.9.4:

Das Programm

```
program testread1 (input, output);
var c : char;
begin
  read (c);
  writeln (c);
end.
```

würde, ohne eine Benutzereingabe zu ermöglichen, das Zeilenendezeichen (Leerzeichen) einlesen und ausgeben.

Bei der Eingabe von Zeichen muß also dem Zeilenendezeichen, im Gegensatz zur Eingabe von *integer*- oder *real*-Zahlen, bei der es wie ein Leerzeichen überlesen wird, besondere Beachtung geschenkt werden. Geschickte Anwendung von *readln* oder *get* (vgl. Abschnitt 2.5.2) schafft Abhilfe.

Durch die Verwendung einer Prozedur *read_char*, vereinbart gemäß

```
procedure read_char (var f: text; var c: char);
begin
  if eoln (f) then
    readln (f);
  read (f, c);
end;
```

kann ein Zeichen (ungleich dem Zeilenendezeichen) ohne weitere *eoln*-Abfragen und ohne die oben genannten unerwarteten Effekte von einer beliebigen Textdatei eingelesen werden.

Beispiel 2.9.5:

Das Programm

```
program testread2 (input, output);
var c : char;
procedure read_char ...
{ Vereinbarung wie oben }
...
begin
  read_char (input, c);
  writeln (c);
end.
```

erlaubt dem Benutzer die Eingabe eines Zeichens, das anschließend wieder ausgegeben wird.

PASCAL–XSC

Die Besonderheiten von PASCAL bezüglich der Eingabe von Zeichen in Verbindung mit dem Zeilenendezeichen gelten in PASCAL–XSC auch für die Eingabe von dynamischen Zeichenketten (Strings). Die beiden folgenden Tabellen sollen diesen Sachverhalt anhand einiger Fallbeispiele verdeutlichen. In den folgenden Tabellen stehen $\hookleftarrow$ für die Beendigung der Eingabezeile durch die *Return*-Taste oder für das Zeilenendezeichen in einer Datei und $\sqcup$ für ein Leerzeichen.

PASCAL–XSC

Eingabe über Konsole:

var S5 : string[5];
S10 : string[10];

Anweisungen	Eingabe	Ausgabe
read (S5, S10) writeln (S5); writeln (S10);	*nicht möglich*	 $\sqcup$ $\sqcup$
readln (S5, S10) writeln (S5); writeln (S10);	ABCDEFGHIJKLMNO$\hookleftarrow$	 $\sqcup$ $\sqcup$
readln; read (S5, S10); writeln (S5); writeln (S10);	ABCDEFGHIJKLMNO$\hookleftarrow$	 ABCDE FGHIJKLMNO
readln; readln (S5, S10); writeln (S5); writeln (S10);	ABCDE$\hookleftarrow$ FGHIJKLMNO$\hookleftarrow$	 ABCDE $\sqcup$
readln; read (S5); readln; read (S10); writeln (S5); writeln (S10);	ABCDE$\hookleftarrow$ FGHIJKLMNO$\hookleftarrow$	 ABCDE FGHIJKLMNO

Eingabe über Datei:

var f : text;

Anweisungen	Dateinhalt	Ausgabe
read (f, S5, S10); writeln (S5); writeln (S10);	ABCDEFGHIJKLMNO	 ABCDE FGHIJKLMNO
read (f, S5, S10); writeln (S5); writeln (S10);	ABCDE$\hookleftarrow$ FGHIJKLMNO	 ABCDE $\sqcup$
readln (f, S5); readln (f, S10); writeln (S5); writeln (S10);	ABCDE$\hookleftarrow$ FGHIJKLMNO	 ABCDE FGHIJKLMNO

PASCAL–XSC

Auch für den Typ *string* läßt sich durch geschickte Anwendung von *readln* das unerwartete Einlesen des Leer- bzw. Zeilenendezeichens vermeiden. Dies kann z. B. mittels einer Überladung der Prozedur *read* (vgl. Abschnitt 2.7.11) geschehen:

```
procedure read (var f : text; var s: string);
var c : char;
begin
  if eoln (f) then
    readln (f);
  s := '';
  while not eoln (f) do
  begin
    read (f, c);
    s := s + c;
  end
end;
```

Für eine beliebige Textdatei *f* und eine *string*-Variable *s* kann diese Prozedur nun, aufgrund des in Abschnitt 2.7.11 beschriebenen Überladungskonzepts, in den Formen

read (s);   read (input, s);   read (f, s);

zur zeilenweisen Eingabe von dynamischen Zeichenketten verwendet werden.

Beispiel 2.9.6:

Mit Hilfe eines PASCAL–XSC Programms soll ein von einer Datei einzulesender Text aus Kleinbuchstaben, der anstelle von Umlauten die Zeichenfolgen `ae`, `oe` und `ue` enthält, in eine vom Text-System LaTeX zu interpretierende Form gebracht werden. Dazu muß im Text ersetzt werden:

`ae` durch `"a`

`oe` durch `"o`

`ue` durch `"u`

Sonderfälle wie `aee` sollen der Übersicht halber nicht berücksichtigt werden. Außerdem soll das Wort `PASCAL`, das mehrmals im Text vorkommt, durch `Pascal` ersetzt und für den Fettdruck gekennzeichnet werden. Letzteres geschieht durch die Umrahmung des Wortes gemäß:

```
{\bf Pascal}
```

PASCAL–XSC

Das folgende PASCAL–XSC Programm liest den zu editierenden Text zeilenweise vom Textfile *texin.txt* ein und gibt diesen, nach der Bearbeitung mit den PASCAL–XSC Stringfunktionen, in das Textfile *texout.txt* aus.

```
program Umlaute (output, indat, outdat);

operator ** (zeile, umlaut: string) res : string;
   {Ersetzt in zeile jedes Vorkommen von umlaut}
   {durch die entsprechende TeX-Sequenz}
   var
      p : integer;
   begin
      p := pos (umlaut, zeile);
      while (p > 0) do
      begin
         zeile[p] := '"';
         zeile[p+1] := umlaut[1];
         p := pos (umlaut, zeile);
      end;
      res := zeile;
   end;

var
   zeile, help1, help2 : string;
   indat, outdat : text;
   laenge, position : integer;

begin
   reset (indat, 'texin.txt');
   rewrite (outdat, 'texout.txt');

   while not eof (indat) do
   begin
      readln (indat, zeile);
      zeile := zeile ** 'ae';
      zeile := zeile ** 'oe';
      zeile := zeile ** 'ue';
      laenge := length (zeile);
      position := pos('PASCAL',zeile);
```

PASCAL–XSC

```
        while (position > 0) do
        begin
          help1:= substring (zeile, 1, position-1) + '{\bf Pascal}';
          help2:= substring (zeile, position+6, laenge−position−5);
          zeile:= help1 + help2;
          laenge:= length (zeile);
          position:= pos ('PASCAL', zeile);
        end;
        writeln (outdat, zeile);
      end;
    end.
```

2.10 Handhabung von dynamischen Feldern

PASCAL–XSC

Bereits in Abschnitt 2.3.2 wurde darauf hingewiesen, daß echte Dynamik in Verbindung mit der Vereinbarung von Feldern nur innerhalb von Prozeduren und Funktionen auftreten kann. Zur Vereinbarung von Variablen eines dynamischen Typs werden dort in den Indexausdrücken globale Größen oder formale Argumente verwendet. Im Hauptprogramm können dagegen in den Ausdrücken für die Indexgrenzen nur Konstanten, importierte Variablen oder zum Vereinbarungszeitpunkt auswertbare Ausdrücke auftreten.

Während ein erfahrener Programmierer unter Verwendung eines speziellen Modulinitialisierungsteils oder durch Funktionsaufrufe für die Indexgrenzen die volle Dynamik auch im Hauptprogramm möglich machen könnte, wird in diesem Abschnitt die übliche Vorgehensweise zur Handhabung dynamischer Felder beschrieben. Diese besteht darin, daß das eigentliche Hauptprogramm, in dem mit dynamischen Feldern gearbeitet werden soll, in eine Prozedur oder Funktion verlegt wird und der Rumpf des Programms nur noch aus dem Einlesen der für die Indexgrenzenberechnung notwendigen Werte und dem Aufruf dieser „Hauptprozedur“ oder „Hauptfunktion“ besteht.

Schematisch sieht demnach ein PASCAL–XSC Programm, das dynamische Felder benutzt, wie folgt aus:

```
program dynprog (input,output);
type
  dyntyp = dynamic array [*] of typ;
  { Weitere Vereinbarungen }
  ...
var
  low, upp: integer;
  { Weitere Vereinbarungen }
  ...
procedure haupt (low, upp: integer);
  var
    a, b, c: dyntyp [low..upp];
    { Weitere Vereinbarungen }
    ...
  begin
    { In die Prozedur verlegtes Hauptprogramm }
    ...
  end;
```

PASCAL–XSC

```
begin { Neues Hauptprogramm }
   read (low,upp);
   haupt (low,upp);
end.
```

Im neuen Hauptprogramm kann der Aufruf der Prozedur *haupt* auch in einer Schleife erfolgen, in der jeweils neue Indexgrenzen *low* und *upp* eingelesen werden. Dies kann z. B. bei der Verwendung eines Algorithmus sinnvoll sein, der durch die Veränderung der Dimension der verwendeten dynamischen Felder ein erzieltes Ergebnis solange verbessert, bis eine bestimmte Genauigkeit erreicht ist.

Beispiel 2.10.1:

```
program langzahl (input,output);
type
   lang = dynamic array [*] of real;
var
   len: integer;
   ...
function genau_genug (len: integer) : boolean;
   var
      lz1, lz2, lz3: lang [1..len];
      erg: real;
      ...
   begin
      { Algorithmus }
      ...
      writeln ('Ergebnis mit Länge ', len:1, ': ', erg);
      if { Genauigkeit für erg erreicht } then
         genau_genug := true
      else
         genau_genug := false;
   end;

begin
   repeat
      write ('Länge der Langzahlen: ');
      read (len);
   until genau_genug (len);
end.
```

PASCAL–XSC

Als abschließendes Beispiel zur Handhabung dynamischer Felder geben wir ein Programm an, das für quadratische und rechteckige Matrizen beliebiger Dimension die Transponierte berechnet und ausgibt.

Beispiel 2.10.2:

```
program transpo (input,output);

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

function transp (var a: matrix) :
   matrix [lbound(a,2)..ubound(a,2) , lbound(a,1)..ubound(a,1)];

   var i, j: integer;

   begin
      for i:=lbound (a,1) to ubound (a,1) do
         for j:=lbound (a,2) to ubound (a,2) do
            transp[j,i] := a[i,j];
   end;

procedure haupt (z, s: integer);

   var
      i, j: integer;
      A: matrix [1..z,1..s];
      T: matrix [1..s,1..z];

   begin
      writeln ('Matrixelemente von A zeilenweise:');
      for i:=1 to z do
         for j:=1 to s do
            read(A[i,j]);
      writeln ('Transponierte von A:');
      T:= transp(A);
      for i:=1 to s do
      begin
         for j:=1 to z do
            write (T[i,j]);
         writeln;
      end;
   end;
```

PASCAL–XSC

```
var z, s: integer;

begin
  writeln('Größe von A eingeben:');
  write('Zeilenanzahl: '); read (z);
  write('Spaltenanzahl: '); read (s);
  while (z > 0) and (s > 0) do
  begin
    haupt (z,s);
    writeln('Größe von A eingeben:');
    write('Zeilenanzahl: '); read (z);
    write('Spaltenanzahl: '); read (s);
  end
end.
```

Kapitel 3

Die Arithmetikmodule

Numerische Methoden erfordern nicht nur Berechnungen mit reellen Zahlen, sondern auch mit komplexen Größen, Intervallen, komplexen Intervallen sowie mit Vektoren und Matrizen über diesen (siehe z. B. [1], [2], [17] oder [29]). Für alle diese Erfordernisse stellt PASCAL–XSC die entsprechenden Datentypen mit den zugehörigen Operatoren und Standardfunktionen zur Verfügung.

Alle arithmetischen Operationen erfüllen wie die in Abschnitt 2.4.1.2 beschriebenen reellen Gleitkommaoperationen die Forderung nach *maximaler Genauigkeit,* d. h. das jeweilige Ergebnis wird komponentenweise auf 1 Ulp genau berechnet.

In PASCAL–XSC steht somit für die zusätzlichen numerischen Datentypen

complex	für	komplexe Zahlen
interval	für	reelle Intervalle
cinterval	für	komplexe Intervalle
rvector	für	reelle Vektoren
cvector	für	komplexe Vektoren
ivector	für	Intervallvektoren
civector	für	komplexe Intervallvektoren
rmatrix	für	reelle Matrizen
cmatrix	für	komplexe Matrizen
imatrix	für	Intervallmatrizen
cimatrix	für	komplexe Intervallmatrizen

ein vollständiges Ausdruckskonzept zur Verfügung. Dabei beschränkt sich dieses Ausdruckskonzept nicht auf Operanden gleichen Typs, sondern es ist erweitert auf fast alle mathematisch sinnvollen Verknüpfungen verschiedener Operandentypen. Dadurch müssen insgesamt weit über 1000 arithmetische Operatoren zur Verfügung gestellt werden. Weiterhin bietet PASCAL–XSC mit einer beinahe ebenso großen Zahl von Vergleichsoperatoren die Möglichkeit, logische Ausdrücke mit den oben genannten arithmetischen Datentypen zu formulieren.

Diese Vielzahl von Operatoren und auch Funktionen ermöglicht es nun, die in vielen Anwendungsbereichen der Ingenieur- und Naturwissenschaften vorkommenden Berechnungen in einer übersichtlichen Form in Programmcode zu überführen. Zumeist können die theoretischen Formeln bzw. Algorithmen in der üblichen mathematischen Notation als Programmtext formuliert werden, was zusätzlich durch einige Überladungen des Zuweisungsoperators := unterstützt wird.

Die nachfolgende Tabelle 1 gibt eine Übersicht über die vordefinierten arithmetischen Operatoren für die arithmetischen Standardtypen von PASCAL–XSC.

linker Operand \ rechter Operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
[1])	$+,-$	$+,-$	$+,-$	$+,-$	$+,-$	$+,-$
integer real complex	$\circ, \circ\!<, \circ\!>,$[2]) $+*$	$+,-,*,/,$ $+*$	$*, *\!<, *\!>$	$*$	$*, *\!<, *\!>$	$*$
interval cinterval	$+,-,*,/,$ $+*$	$+,-,*,/,$ $+*, **$	$*$	$*$	$*$	$*$
rvector cvector	$*, *\!<, *\!>,$ $/, /\!<, /\!>$	$*, /$	$\circ, \circ\!<, \circ\!>,$[3]) $+*$	$+,-,*,$[4]) $+*$		
ivector civector	$*, /$	$*, /$	$+,-,*,$[4]) $+*$	$+,-,*,$[4]) $+*, **$		
rmatrix cmatrix	$*, *\!<, *\!>,$ $/, /\!<, /\!>$	$*, /$	$*, *\!<, *\!>$	$*$	$\circ, \circ\!<, \circ\!>,$[3]) $+*$	$+,-,*,$[4]) $+*$
imatrix cimatrix	$*, /$	$*, /$	$*$	$*$	$+,-,*,$[4]) $+*$	$+,-,*,$[4]) $+*, **$

[1]) Die Operatoren dieser Zeile sind monadisch (es gibt keinen linken Operanden).

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

[3]) $\circ \in \{+,-,*\}$, wobei der $*$-Operator für das Skalar- bzw. Matrixprodukt steht.

[4]) Der $*$-Operator steht für das Skalar- bzw. das Matrixprodukt.

$+*$: Intervall-Hülle

$**$: Schnittmenge

Tabelle 1: Vordefinierte arithmetische Operatoren

Bemerkung: Der mit [2]) gekennzeichnete Block in Tabelle 1 enthält auch die im Standard vorhandenen Operationen für *real*- und *integer*-Operanden. Die Operationen $*$ (Skalarprodukt) bzw. $+*$ (Intervall-Hülle) und $**$ (Schnittmenge) werden in den entsprechenden Matrix/Vektor-Modulen bzw. Intervallmodulen erläutert.

Tabelle 2 gibt einen Überblick über die darüber hinaus zur Verfügung stehenden Vergleichs-Operatoren für die arithmetischen Standardtypen von PASCAL–XSC.

rechter Operand / linker Operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
integer real complex	=, <>, <=, <, >=, >	**in** =, <>				
interval cinterval	=, <>	**in**, ><, [1]) =, <>, <=, <, >=, >				
rvector cvector			=, <>, <=, <, >=, >	**in** =, <>		
ivector civector			=, <>	**in**, ><, [1]) =, <>, <=, <, >=, >		
rmatrix cmatrix					=, <>, <=, <, >=, >	**in** =, <>
imatrix cimatrix					=, <>	**in**, ><, [1]) =, <>, <=, <, >=, >

[1]) Die Operatoren <= und < stehen für „Teilmenge von“ und „echte Teilmenge von“, >= und > sind die entsprechenden Operatoren für die Obermengenbeziehung.

>< : Disjunktheitstest für Intervalle

in : Test ob Punktgröße in einer Intervallgröße liegt oder
Test ob Intervallgröße echt im Innern einer Intervallgröße liegt

Tabelle 2: Vordefinierte Vergleichsoperatoren

Die große Anzahl von Operatoren wirft die Frage auf, in welcher Weise diese in der Sprache verfügbar gemacht werden. Das Modulkonzept von PASCAL–XSC bietet die Voraussetzungen, dies in Form von entsprechenden Arithmetikmodulen zu realisieren. Neben den aufgeführten Operatoren wird in diesem Modulen auch zu jedem Datentyp ein Satz von Standardfunktionen zur Verfügung gestellt. Für die Typen *complex*, *interval* und *cinterval* umfaßt dieser auch alle vom Datentyp *real* bekannten mathematischen Standardfunktionen.

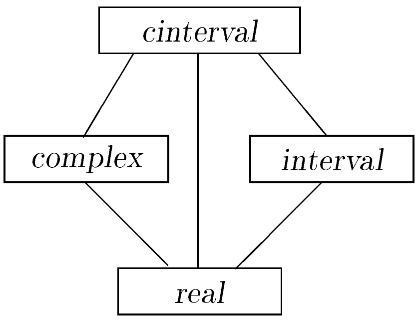
Damit stehen insgesamt die folgenden Module zur Verfügung:

C_ARI	Komplexe Arithmetik
I_ARI	Intervallarithmetik
CI_ARI	Komplexe Intervallarithmetik
MV_ARI	Reelle Matrix/Vektor-Arithmetik
MVC_ARI	Komplexe Matrix/Vektor-Arithmetik
MVI_ARI	Intervall-Matrix/Vektor-Arithmetik
MVCI_ARI	Komplexe Intervall-Matrix/Vektor-Arithmetik

Nachfolgend werden die einzelnen Module beschrieben, indem die Datentypen, die Operatoren, die Transferfunktionen, die Überladungen von :=, die Standardfunktionen und die Ein-/Ausgabeprozeduren in dieser Reihenfolge systematisch erläutert werden. Für die Ein-/Ausgabeprozeduren gilt das in Abschnitt 2.7.11 beschriebene Überladungsprinzip für *read* und *write*, d. h. sie können mit optionalem Fileparameter und beliebiger Argumentanzahl aufgerufen werden. Die Beschreibung dieser Prozeduren beschränkt sich deshalb zumeist auf die Erläuterung der Eingabesyntax.

Definition der arithmetischen Operatoren

Der Ergebnistyp der skalaren arithmetischen Operationen ist im mathematischen Sinne definiert gemäß der folgenden Typhierarchie:



Das Ergebnis ist also stets der niedrigste Typ, der beide Operandentypen enthält.

Bei den Matrix/Vektor-Operationen ergibt sich die Struktur des Ergebnistyps aus den Strukturen der Operanden gemäß nachfolgender Übersicht:

$$\begin{array}{ccccc} v & + & v & = & v \\ m & + & m & = & m \\ v & - & v & = & v \\ m & - & m & = & m \\ v & / & s & = & v \\ m & / & s & = & m \end{array} \qquad \begin{array}{ccccc} v & * & v & = & s \\ s & * & v & = & v \\ v & * & s & = & v \\ m & * & v & = & v \\ s & * & m & = & m \\ m & * & s & = & m \\ m & * & m & = & m \end{array}$$

Struktur des Ergebnistyps bei Matrix/Vektor-Operationen
s = Skalar, v = Vektor, m = Matrix

Der Ergebnistyp ergibt sich dann ebenfalls gemäß obiger Typhierarchie in Abhängigkeit der beiden Komponententypen.

Alle Matrix/Vektor-Operationen setzen dabei nur voraus, daß die Anzahlen der zusammentreffenden Komponenten der Operanden gleich sind, d. h. daß die entsprechenden Indexbereiche die gleiche Länge haben. Die Indexbereiche selbst können dagegen durchaus verschieden sein, wie etwa in

```
var
  p : real;
  a : rvector[1..10];
  b : cvector[11..20];
...
p := a * b;
```

Bei der Definition der Operatoren in den nachfolgenden Abschnitten wird jedoch der einfacheren Beschreibung wegen davon ausgegangen, daß die auftretenden Indexbereiche identisch sind. Das Skalarprodukt $p = a * b$ wird zum Beispiel beschrieben durch

```
#* (for i:= lb(a) to ub(a) sum
      (a[i] * b[i]) )
```

ist aber tatsächlich implementiert als

```
#* (for i:= lb(a) to ub(a) sum
      (a[i] * b[i + lb(b) - lb(a)]) )
```

Definition der Vergleichsoperatoren

Als Basis für die Vergleichsoperatoren in den Modulen dient der Satz der für den Standardtyp *real* vorhandenen Vergleichsoperatoren, mit deren Hilfe die Operatoren <= und = für einen strukturierten numerischen Datentyp SNDT (Strukturierter Numerischer Daten-Typ) definiert sind. Dabei ist der Vergleichsoperator = stets so implementiert, daß er genau dann *true* liefert, wenn für alle Komponenten des SNDTs die Gleichheit erfüllt ist. Die vom jeweiligen Datentyp abhängige Definition des Operators <= wird in den entsprechenden Abschnitten zu den einzelnen Modulen angegeben.

Alle weiteren Vergleichsoperatoren für Elemente a, $b \in$ SNDT werden dann gemäß (VD) wie folgt definiert:

$$
\text{(VD)}\quad
\begin{array}{ccccl}
\mathrm{a} & <> & \mathrm{b} & := & \textbf{not}\ (\mathrm{a} = \mathrm{b}) \\
\mathrm{a} & < & \mathrm{b} & := & (\mathrm{a} <= \mathrm{b})\ \textbf{and}\ (\mathrm{a} <> \mathrm{b}) \\
\mathrm{a} & > & \mathrm{b} & := & \mathrm{b} < \mathrm{a} \\
\mathrm{a} & >= & \mathrm{b} & := & \mathrm{b} <= \mathrm{a}
\end{array}
$$

Überladungen des Zuweisungsoperators

In Form von Überladungen des Zuweisungsoperators werden in den Arithmetikmodulen einige grundlegende, mathematisch sinnvolle Typwandlungen bzw. Initialisierungen zur Verfügung gestellt. Dabei sind die Typwandlungen grundsätzlich im Sinne der mathematischen Einbettung (wie z. B. der reellen Zahlen in die komplexen Zahlen), also *werterhaltend* realisiert. Initialisierungen in Form von Zuweisungen skalarer Typen an Vektor- oder Matrix-Typen sind stets komponentenweise definiert (jede Komponente erhält denselben Wert). Rundungs- oder Konversionsfehler finden bei diesen Zuweisungen *nicht* statt.

Bei Verwendung von Literalkonstanten auf der rechten Seite der Zuweisung, gilt es jedoch auch hier, die in Abschnitt 2.3.1 beschriebene Problematik der Konvertierung zu beachten. Eine reelle Konstante wird dementsprechend bereits *vor* der Ausführung des Zuweisungsoperators ins interne *real*-Format konvertiert.

Genauigkeit der Standardfunktionen

Alle komplexen Standardfunktionen liefern Ergebnisse, die auf 1 Ulp genau sind.

Die Intervall-Standardfunktionen berechnen stets ein Gleitkommaintervall, welches das exakte Ergebnisintervall enthält. Dabei wird in den meisten Fällen das kleinste einschließende Gleitkommaintervall berechnet. Aus implementierungstechnischen Gründen können in wenigen Spezialfällen die Endpunkte des berechneten Intervalls von den exakten Endpunkten schlimmstenfalls um 2 Ulp abweichen.

Für die komplexen Intervall-Standardfunktionen gilt dies jeweils für den Real- und Imaginärteil.

3.1 Das Modul C_ARI
Komplexe Arithmetik

In diesem Modul werden die für das Rechnen mit komplexen Zahlen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentyp

Der Datentyp *complex* ist entsprechend der Definition

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

im Sprachkern von PASCAL–XSC enthalten. Diesem liegt die kartesische Darstellung einer komplexen Zahl z in der Form

$$z = x + iy$$

zugrunde, wobei x den Realteil und y den Imaginärteil von z darstellen.

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 zur Verfügung. Dabei ist die Rundung jeweils komponentenweise zu verstehen.

Die Vergleichsoperatoren $=, <>, <, <=, >, >=$ sind auf der Basis von $=$ und $<=$ wie eingangs erwähnt gemäß (VD) definiert (vgl. Seite 136), wobei für *a*, *b* vom Typ *complex* gilt:

a $<=$ b $\iff$ (a.re $<=$ b.re) **and** (a.im $<=$ b.im)

Auch Vergleiche mit einem *integer*- oder *real*-Operanden sind zulässig.

rechter Operand / linker Operand	integer real	complex
monadisch		$+, -$
integer real		$\circ$ $\vee$
complex	$\circ$ $\vee$	$\circ$ $\vee$

Die Operatoren des Moduls C_ARI

$$\circ \in \{+, +<, +>, -, -<, ->, *, *<, *>, /, /<, />\}$$

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

Transferfunktionen

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

Funktion	Ergebnistyp	Bedeutung
compl (r1,r2)	*complex*	Komplexe Zahl mit Realteil *r1* und Imaginärteil *r2*
compl (r)	*complex*	Rein reelle komplexe Zahl mit Realteil *r* und Imaginärteil 0
re (c)	*real*	Realteil von *c*
im (c)	*real*	Imaginärteil von *c*

r, r1, r2 = *real*-Ausdruck, c = *complex*-Ausdruck

Beispiel 3.1.1:

Die imaginäre Einheit i wird in PASCAL–XSC durch den Ausdruck

compl (0,1)

erzeugt.

Überladungen des Zuweisungsoperators

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

Zuweisung	Bedeutung
c := r	c := compl (r)

c = *complex*-**Variable**
r = *real*-Ausdruck

Standardfunktionen

Alle für *real*-Argumente verfügbaren mathematischen Standardfunktionen von PASCAL–XSC werden für komplexe Argumente bereitgestellt. Darüber hinaus sind Funktionen für die Berechnung des Winkelarguments φ der Exponentialdarstellung $z = r \cdot e^{i\varphi}$ einer komplexen Zahl z und für die Konjugation (Spiegelung an der reellen Achse) enthalten.

Funktion	Ergebnistyp	Bedeutung	
sqr (c)	*complex*	$c^2 = c \cdot c$	Quadrat
sqrt (c)	*complex*	$\sqrt{c}$	Quadratwurzel (Realteil > 0)
exp (c)	*complex*	e^c	Exponentialfunktion
exp2 (c)	*complex*	2^c	Exponentiation zur Basis 2
exp10 (c)	*complex*	10^c	Exponentiation zur Basis 10
ln (c)	*complex*	$\ln(c)$	natürlicher Logarithmus
log2 (c)	*complex*	$\log_2(c)$	Logarithmus zur Basis 2
log10 (c)	*complex*	$\log_{10}(c)$	Logarithmus zur Basis 10
sin (c)	*complex*	$\sin(c)$	Sinus
cos (c)	*complex*	$\cos(c)$	Kosinus
tan (c)	*complex*	$\tan(c)$	Tangens
cot (c)	*complex*	$\cot(c)$	Kotangens
arcsin (c)	*complex*	$\arcsin(c)$	Arkussinus
arccos (c)	*complex*	$\arccos(c)$	Arkuskosinus
arctan (c)	*complex*	$\arctan(c)$	Arkustangens
arccot (c)	*complex*	$\mathrm{arccot}(c)$	Arkuskotangens
sinh (c)	*complex*	$\sinh(c)$	Hyperbolischer Sinus
cosh (c)	*complex*	$\cosh(c)$	Hyperbolischer Kosinus
tanh (c)	*complex*	$\tanh(c)$	Hyperbolischer Tangens
coth (c)	*complex*	$\coth(c)$	Hyperbolischer Kotangens
arsinh (c)	*complex*	$\mathrm{arsinh}(c)$	Areasinus
arcosh (c)	*complex*	$\mathrm{arcosh}(c)$	Areakosinus
artanh (c)	*complex*	$\mathrm{artanh}(c)$	Areatangens
arcoth (c)	*complex*	$\mathrm{arcoth}(c)$	Areakotangens
conj (c)	*complex*	$\bar{c} = x - iy$	Konjugation von $c = x + iy$
arg (c)	*real*	φ	Argument von $c = r \cdot e^{i\varphi}$
abs (c)	*real*	$r = \sqrt{x^2 + y^2}$	Absolutbetrag von $c = r \cdot e^{i\varphi} = x + iy$

c, c1, c2 = *complex*-Ausdruck

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 $c = x + iy$ muß in der Form

(x, y)

oder in der Form

x

erfolgen. Im zweiten Fall wird der Imaginärteil y automatisch auf 0 gesetzt. Die *real*-Konstanten x und y werden dabei zur jeweils nächstliegenden Gleitkommazahl gerundet.

Die Ausgabe eines komplexen Wertes erfolgt stets nächstliegend gerundet in der Form

(x, y)

in einem implementierungsabhängigen Standardformat für die *real*-Größen x und y.

Beispiel 3.1.2:

Sei c vom Typ *complex*, dann wird mit den Anweisungen

```
read (c);
writeln (c);
```

und der Eingabe

-1.23456789

der *complex*-Wert

(−1.234567890000E+00, 0.000000000000E+00)

ausgegeben, je nach Implementierung mit anderer *real*-Darstellung.

3.2 Das Modul I_ARI
Intervallarithmetik

In diesem Modul werden die für das Rechnen mit reellen Intervallen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentyp

Der Datentyp *interval* ist entsprechend der Definition

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

im Sprachkern von PASCAL–XSC enthalten. Diesem liegt die übliche Darstellung eines reellen Intervalls x in der Form

$$x = [x_{\text{inf}}, x_{\text{sup}}]$$

zugrunde, welches die Menge $\{y \in I\!R \mid x_{\text{inf}} \leq y \leq x_{\text{sup}}\}$ repräsentiert. Dabei stehen *inf* für Infimum (untere Grenze) und *sup* für Supremum (obere Grenze).

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

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

Die Operatoren sind auf der Basis von $=$ und $<=$ wie eingangs erwähnt gemäß (VD) definiert (vgl. Seite 136), wobei für *x, y* vom Typ *interval* gilt:

$$\text{x} \;<=\; \text{y} \iff (\text{x.inf} \;>=\; \text{y.inf}) \;\textbf{and}\; (\text{x.sup} \;<=\; \text{y.sup})$$

Zusätzlich stehen die Operatoren **in** für die Relation „liegt in“ zwischen einem *real*- und einem *interval*-Operanden bzw. „enthalten im Innern“ zwischen zwei *interval*-Operanden sowie $><$ für den Test auf Disjunktheit zweier reeller Intervalle zur Verfügung. Dabei heißen zwei Intervalle x, y disjunkt, wenn gilt $x \cap y = \emptyset$ (leere Menge). Es gilt

$$\text{x} \;\textbf{in}\; \text{y} \iff (\text{x.inf} \;>\; \text{y.inf}) \;\textbf{and}\; (\text{x.sup} \;<\; \text{y.sup}).$$

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.

linker Operand \ rechter Operand	integer real	interval
monadisch		$+, -$
integer real	$+*$	$\diamond$ **in**, $=, <>$ $+*$
interval	$\diamond$ $=, <>$ $+*$	$\diamond$ **in**, $\vee, ><$ $+*, **$

Die Operatoren des Moduls I_ARI

$$\diamond \in \{+, -, *, /\}$$

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

Beispiel 3.2.1:

Seien a,b vom Typ *interval* mit

$$\begin{array}{lcl} \mathrm{a} & = & [-1{,}3] \\ \mathrm{b} & = & [3{,}4] \end{array}$$

dann liefern beispielsweise die Operatoren $+$, $-$, $*$, $><$, $+*$ und $**$ die folgenden Ergebnisse:

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

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* (*)
intval (r)	*interval*	Punktintervall mit *inf* = *sup* = *r*
inf (i)	*real*	Untergrenze von *i*
sup (i)	*real*	Obergrenze von *i*

r, r1, r2 = *real*-Ausdruck, i = *interval*-Ausdruck

(*): Es muß r1 <= r2 gelten, sonst tritt ein Laufzeitfehler auf.

Überladungen des Zuweisungsoperators

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

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

i = *interval*-**Variable**
r = *real*-Ausdruck

Standardfunktionen

Alle für *real*-Argumente verfügbaren mathematischen Standardfunktionen von PASCAL–XSC werden für Intervall-Argumente i bereitgestellt. Für diese Intervallfunktionen F gilt stets $F(i) \supseteq f(i) = \{f(r) : r \in i\}$. Darüber hinaus sind Funktionen für die Berechnung des Mittelpunktes und des Durchmessers von Intervallen verfügbar. Im Zusammenhang mit Einschließungsmethoden wird die Funktion *blow* für die sogenannte *Epsilonaufblähung* (vgl. [34]) zur Verfügung gestellt.

Funktion	Ergebnistyp	Bedeutung	
sqr (i)	*interval*	i^2	Intervallquadrat
sqrt (i)	*interval*	$\sqrt{i}$	Quadratwurzel
exp (i)	*interval*	e^i	Exponentialfunktion
exp2 (i)	*interval*	2^i	Exponentiation zur Basis 2
exp10 (i)	*interval*	10^i	Exponentiation zur Basis 10
ln (i)	*interval*	$\ln(i)$	natürlicher Logarithmus

i = *interval*-Ausdruck

Funktion	Ergebnistyp	Bedeutung	
log2 (i)	*interval*	$\log_2(i)$	Logarithmus zur Basis 2
log10 (i)	*interval*	$\log_{10}(i)$	Logarithmus zur Basis 10
sin (i)	*interval*	sin (i)	Sinus
cos (i)	*interval*	cos (i)	Kosinus
tan (i)	*interval*	tan (i)	Tangens
cot (i)	*interval*	cot (i)	Kotangens
arcsin (i)	*interval*	arcsin (i)	Arkussinus
arccos (i)	*interval*	arccos (i)	Arkuskosinus
arctan (i)	*interval*	arctan (i)	Arkustangens
arctan2 (i1,i2)	*interval*	arctan $(i1/i2)$	Arkustangens
arccot (i)	*interval*	arccot (i)	Arkuskotangens
sinh (i)	*interval*	sinh (i)	Hyperbolischer Sinus
cosh (i)	*interval*	cosh (i)	Hyperbolischer Kosinus
tanh (i)	*interval*	tanh (i)	Hyperbolischer Tangens
coth (i)	*interval*	coth (i)	Hyperbolischer Kotangens
arsinh (i)	*interval*	arsinh (i)	Areasinus
arcosh (i)	*interval*	arcosh (i)	Areakosinus
artanh (i)	*interval*	artanh (i)	Areatangens
arcoth (i)	*interval*	arcoth (i)	Areakotangens
abs (i)	*interval*	$\|i\| = \{\|r\| : r \in i\}$	Absolutbetrag
mid (i)	*real*	m =#∗(0.5∗inf(i) + 0.5∗sup(i))	Mittelpunkt von i
diam (i)	*real*	d = sup(i) −> inf(i)	Durchmesser von i
blow (i,r)	*interval*	‡	Epsilonaufblähung

i, i1, i2 = *interval*-Ausdruck r = *real*-Ausdruck

‡ : y := (1 + r) ∗ i − r ∗ i;
blow := intval (pred(inf(y)),succ(sup(y)));

Beispiel 3.2.2:

Sei *a,b* vom Typ *interval* erzeugt durch

a := intval (−1,3)
b := intval (2)

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

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

Ein-/Ausgabeprozeduren

Es stehen die gewohnten 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 $i = [x, y]$ muß in der Form

$$[x, y]$$

oder in der Form

$$x$$

erfolgen. Im ersten Fall wird der eingegebene Wert x zur nächstkleineren, der Wert y zur nächstgrößeren Gleitkommazahl gerundet (Rundung zum kleinsten umfassenden Maschinenintervall). Der zweite Fall dient zur vereinfachten Eingabe des Punktintervalls $i = [x, x]$. Falls x nicht exakt darstellbar ist, wird das kleinste, x enthaltende Intervall erzeugt.

Die Ausgabe eines Intervalls erfolgt stets mit Rundung nach außen (d. h. x nach unten, y nach oben) in der Form

$$[x, y]$$

mit einem implementierungsabhängigen Standardformat für die *real*-Größen x und y.

Beispiel 3.2.3:

Sei *int* vom Typ *interval*, dann wird mit den Anweisungen

```
read (int);
writeln (int);
```

und der Eingabe

```
0.245
```

das Punktintervall

```
[ 2.45E−01, 2.45E−01]
```

ausgegeben, je nach Implementierung mit anderer *real*-Darstellung.

3.3 Das Modul CI_ARI
Komplexe Intervallarithmetik

In diesem Modul werden die für das Rechnen mit komplexen Intervallen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentyp

Der Datentyp *cinterval* ist entsprechend der Definition

type cinterval = **record** re, im : interval **end**;

im Sprachkern von PASCAL–XSC enthalten. Diesem liegt die Darstellung eines komplexen Intervalls z in der Form

$$z = [x_{\text{inf}}, x_{\text{sup}}] + i \cdot [y_{\text{inf}}, y_{\text{sup}}]$$

zugrunde, welches in der komplexen Zahlenebene ein Rechteck darstellt (Rechteckintervall).

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

Die Operatoren sind auf der Basis von $=$ und $<=$ wie eingangs erwähnt gemäß (VD) definiert (vgl. Seite 136), wobei für *v, w* vom Typ *cinterval* gilt:

v <= w $\Longleftrightarrow$ (v.re <= w.re) **and** (v.im <= w.im) .

Die Operatoren auf der rechten Seite der Äquivalenzbeziehung sind dabei die Operatoren für Intervalle vom Typ *interval*.

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 $v \cap w = \emptyset$. Für zwei komplexe Intervalle *v* und *w* gilt

v **in** w $\Longleftrightarrow$ (v.re **in** w.re) **and** (v.im **in** w.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.

linker Operand \ rechter Operand	integer real	complex	interval	cinterval
monadisch				$+, -$
integer real		$+*$		$\diamond$ **in**, $=, <>$ $+*$
complex	$+*$	$+*$	$\diamond$ **in**, $=, <>$ $+*$	$\diamond$ **in**, $=, <>$ $+*$
interval		$\diamond$ $=, <>$ $+*$		$\diamond$ **in**, $\vee$, $><$ $+*, **$
cinterval	$\diamond$ $=, <>$ $+*$	$\diamond$ $=, <>$ $+*$	$\diamond$ $\vee$, $><$ $+*, **$	$\diamond$ **in**, $\vee$, $><$ $+*, **$

Die Operatoren des Moduls CI_ARI

$$\diamond \in \{+, -, *, /\}$$

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

Beispiel 3.3.1:

Sei *ca* vom Typ *cinterval* mit

$$\text{ca} = [-1{,}3] + i\ [3{,}4]$$

dann liefern beispielsweise die Operatoren $+$, $-$ und $*$ die folgenden Ergebnisse:

Ausdruck	Ergebnis
ca $+$ ca	$[-2{,}6] + i\ [6{,}8]$
ca $-$ ca	$[-4{,}4] + i\ [-1{,}1]$
ca $*$ ca	$[-19{,}0] + i\ [-8{,}24]$

Transferfunktionen

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

Funktion	Ergebnistyp	Bedeutung
compl (i1,i2)	*cinterval*	Komplexes Intervall mit Realteil *i1* und Imaginärteil *i2*
compl (r,i)	*cinterval*	Komplexes Intervall mit Realteil *r* und Imaginärteil *i*
compl (i,r)	*cinterval*	Komplexes Intervall mit Realteil *i* und Imaginärteil *r*
compl (i)	*cinterval*	Komplexes Intervall mit Realteil *i* und Imaginärteil 0
intval (c1,c2)	*cinterval*	Komplexes Intervall mit Realteil *[c1.re,c2.re]* und Imaginärteil *[c1.im,c2.im]* $^{(*)}$
intval (r,c)	*cinterval*	Komplexes Intervall mit Realteil *[r,c.re]* und Imaginärteil *[0,c.im]* $^{(*')}$
intval (c,r)	*cinterval*	Komplexes Intervall mit Realteil *[c.re,r]* und Imaginärteil *[c.im,0]* $^{(*'')}$
intval (c)	*cinterval*	Komplexes Punktintervall mit Realteil *[c.re,c.re]* und Imaginärteil *[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)*

r = *real*-Ausdruck, i, i1, i2 = *interval*-Ausdruck,
c, c1, c2 = *complex*-Ausdruck, ci = *cinterval*-Ausdruck

$^{(*)}$: Es muß c1 <= c2 gelten, sonst tritt ein Laufzeitfehler auf.

$^{(*')}$: Es muß r <= c gelten, sonst tritt ein Laufzeitfehler auf.

$^{(*'')}$: Es muß c <= r gelten, sonst tritt ein Laufzeitfehler auf.

Überladungen des Zuweisungsoperators

Die Wandlungen von *real* bzw. *complex* bzw. *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)

ci = *cinterval*-**Variable**, i = *interval*-Ausdruck
c = *complex*-Ausdruck, r = *real*-Ausdruck

Standardfunktionen

Alle für *real*-Argumente verfügbaren mathematischen Standardfunktionen von PASCAL–XSC werden für komplexe Intervall-Argumente *ci* bereitgestellt. Für diese komplexen Intervallfunktionen F gilt stets $F(ci) \supseteq f(ci) = \{f(c) : c \in ci\}$. Darüber hinaus sind Funktionen für die Berechnung des Winkelarguments der Exponentialdarstellung, für die Konjugation sowie für die Berechnung des Mittelpunktes, des Durchmessers und der Epsilonaufblähung von komplexen Intervallen verfügbar.

Funktion	Ergebnistyp	Bedeutung	
sqr (ci)	*cinterval*	ci^2	Quadrat
sqrt (ci)	*cinterval*	$\sqrt{ci}$	Quadratwurzel
exp (ci)	*cinterval*	e^{ci}	Exponentialfunktion
exp2 (ci)	*cinterval*	2^{ci}	Exponentiation zur Basis 2
exp10 (ci)	*cinterval*	10^{ci}	Exponentiation zur Basis 10
ln (ci)	*cinterval*	$\ln (ci)$	natürlicher Logarithmus
log2 (ci)	*cinterval*	$\log_2(ci)$	Logarithmus zur Basis 2
log10 (ci)	*cinterval*	$\log_{10}(ci)$	Logarithmus zur Basis 10
sin (ci)	*cinterval*	$\sin (ci)$	Sinus
cos (ci)	*cinterval*	$\cos (ci)$	Kosinus
tan (ci)	*cinterval*	$\tan (ci)$	Tangens
cot (ci)	*cinterval*	$\cot (ci)$	Kotangens
arcsin (ci)	*cinterval*	$\arcsin (ci)$	Arkussinus
arccos (ci)	*cinterval*	$\arccos (ci)$	Arkuskosinus
arctan (ci)	*cinterval*	$\arctan (ci)$	Arkustangens
arccot (ci)	*cinterval*	$\mathrm{arccot}\ (ci)$	Arkuskotangens

ci, ci1, ci2 = *cinterval*-Ausdruck

Funktion	Ergebnistyp	Bedeutung	
sinh (ci)	*cinterval*	sinh (ci)	Hyperbolischer Sinus
cosh (ci)	*cinterval*	cosh (ci)	Hyperbolischer Kosinus
tanh (ci)	*cinterval*	tanh (ci)	Hyperbolischer Tangens
coth (ci)	*cinterval*	coth (ci)	Hyperbolischer Kotangens
arsinh (ci)	*cinterval*	arsinh (ci)	Areasinus
arcosh (ci)	*cinterval*	arcosh (ci)	Areakosinus
artanh (ci)	*cinterval*	artanh (ci)	Areatangens
arcoth (ci)	*cinterval*	arcoth (ci)	Areakotangens
conj (ci)	*cinterval*	$\overline{ci} = a - ib$	Konjugation von $ci = a + ib$
abs (ci)	*interval*	$j = \sqrt{ci.re^2 + ci.im^2}$	Absolutbetrag von ci
arg (ci)	*interval*	φ	Argumentintervall der Exponentialdarstellung von ci
mid (ci)	*complex*	m	Mittelpunkt von ci
diam (ci)	*real*	d	Durchmesser von ci
blow (ci,r)	*cinterval*	‡	Epsilonaufblähung

ci = *cinterval*-Ausdruck r = *real*-Ausdruck

‡ : blow := compl (blow(ci.re,r),blow(ci.im,r))

Beispiel 3.3.2:

Sei a vom Typ *cinterval* erzeugt durch

a := compl (intval (−1,3), intval (3,4)),

dann liefern beispielsweise die Funktionen *abs* und *sqr* die folgenden Ergebnisse:

Ausdruck	Ergebnis
abs (a)	[3,5]
sqr (a)	[−16,0] + i [−8,24]

Ein-/Ausgabeprozeduren

Es stehen die gewohnten Prozeduren

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

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

Die Eingabe eines komplexen Intervalls $ci = [x, y] + i[v, w]$ muß 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. Dabei werden Real- und Imaginärteil wie in Abschnitt 3.2 beschrieben gerundet.

Die Ausgabe eines komplexen Intervalls erfolgt stets mit der in Abschnitt 3.2 beschriebenen Intervallrundung für Real- und Imaginärteil in der Form

$([x, y], [v, w])$

mit einem implementierungsabhängigen Standardformat für die *real*-Größen x, y, v und w.

Beispiel 3.3.3:

Seien *ci1, ci2* und *ci3* vom Typ *cinterval*, dann werden mit den Anweisungen

```
read (ci1, ci2, ci3);
writeln (ci1);
writeln (ci2);
writeln (ci3);
```

und den Eingaben

```
[4,5]
(8,10)
100
```

die komplexen Intervalle

```
([ 4.0E+00, 5.0E+00],[ 0.0E+00, 0.0E+00])
([ 8.0E+00, 8.0E+00],[ 1.0E+01, 1.0E+01])
([ 1.0E+02, 1.0E+02],[ 0.0E+00, 0.0E+00])
```

ausgegeben, je nach Implementierung mit anderer *real*-Darstellung.

3.4 Das Modul MV_ARI
Reelle Matrix/Vektor-Arithmetik

In diesem Modul werden die für das Rechnen mit reellen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

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.

Operatoren

Viele der aus der Mathematik 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 (auch für gemischte Typen) 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:=lbound(a) **to** ubound(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] ‡

‡: maximal genaues Skalarprodukt

wobei *r, s* vom Typ *real*, *a, b, c* vom Typ *rvector* und *A, B, C* vom Typ *rmatrix* sind. Die Operationen mit gerichteter Rundung nach unten bzw. oben sind entsprechend definiert.

Die Vergleichsoperatoren =, <>, <, <=, >, >= sind auf der Basis von = und <= wie eingangs erwähnt gemäß (VD) realisiert (vgl. Seite 136), wobei für *a,b* vom Typ *rvector* und *A,B* vom Typ *rmatrix* 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 Werte vom Typ *real*.

linker Operand \ rechter Operand	integer real	rvector	rmatrix
monadisch		+, −	+, −
integer real		*, *<, *>	*, *<, *>
rvector	*, *<, *> /, /<, />	∘ ∨	
rmatrix	*, *<, *> /, /<, />	*, *<, *>	∘ ∨

Die Operatoren des Moduls MV_ARI

$$\circ \in \{+,\ +<,\ +>,\ -,\ -<,\ ->,\ *,\ *<,\ *>\}$$

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

Beispiel 3.4.1:

Die im Runge-Kutta-Verfahren zur approximativen Lösung von Anfangswertaufgaben der Form

$$Y' = F(x,Y); \quad Y(x^0) = Y^0;$$

mit

$$Y = \begin{pmatrix} y_1(x) \\ \vdots \\ y_n(x) \end{pmatrix},\ Y' = \begin{pmatrix} y'_1(x) \\ \vdots \\ y'_n(x) \end{pmatrix}$$

und

$$F(x,Y) = \begin{pmatrix} f_1(x,y_1,\ldots,y_n) \\ \vdots \\ f_n(x,y_1,\ldots,y_n) \end{pmatrix}$$

verwendeten Formeln zur Bestimmung einer Näherung der Lösung Y an der Stelle $x+h$ lauten

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

und

$$Y(x+h) = Y(x) + (K_1 + 2K_2 + 2K_3 + K_4)/6.$$

Diese können, nach der Definition der *rvector*-Funktion F und der Vereinbarung von Variablen *k1, k2, k3, k4, Y* vom Typ *rvector* und *h, x* vom Typ *real*, direkt in Programmtext umgesetzt werden durch:

```
k1  :=  h * F (x , Y);
k2  :=  h * F (x + h/2, Y + k1/2);
k3  :=  h * F (x + h/2, Y + k2/2);
k4  :=  h * F (x + h , Y + k3);
Y   :=  Y + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
```

Überladungen des Zuweisungsoperators

Die komponentenweisen Initialisierungen von *rvector*- und *rmatrix*-Variablen werden 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)

r = *real*-Ausdruck, rv = *rvector*-**Variable**, rM = *rmatrix*-**Variable**

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* · *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* · *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]*

n, n1, n2 = *integer*-Ausdruck, v = *rvector*-Ausdruck
M, M1, M2 = *rmatrix*-Ausdruck

Beispiel 3.4.2:

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 := id (A) − R ∗ A

bzw. unter Verwendung eines Lattenkreuzausdrucks (vgl. Abschnitt 2.4.4) durch die Anweisung

D := #∗ (id (A) − R ∗ A)

bestimmt werden, wobei die zweite Form die Defektmatrix mit nur einer einzigen Rundung pro Komponente berechnet.

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 in einem implementierungsabhängigen Standardformat für die reellen Komponenten.

Beispiel 3.4.3:

Durch die Anweisung

read (b, A, x)

können der Vektor b, die Matrix A und der Vektor x hintereinander eingelesen werden.

3.5 Das Modul MVC_ARI
Komplexe Matrix/Vektor-Arithmetik

In diesem Modul werden die für das Rechnen mit komplexen Vektoren und Matrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Vektoren und Matrizen sind entsprechend der Definition

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.

Operatoren

Viele der aus der Mathematik 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 (auch für gemischte Typen) 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:=lbound(a) **to** ubound(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] ‡

‡: maximal genaues Skalarprodukt

wobei *r, s* vom Typ *complex*, *a, b, c* vom Typ *cvector*, und *A, B, C* vom Typ *cmatrix* sind. Die Operationen mit gemischten Operandentypen sowie die Operationen mit gerichteter Rundung nach unten bzw. oben sind entsprechend definiert.

Die Vergleichsoperatoren $=$, $<>$, $<$, $<=$, $>$, $>=$ sind auf der Basis von $=$ und $<=$ wie eingangs erwähnt gemäß (VD) realisiert (vgl. Seite 136), wobei für *a,b* vom Typ *cvector* und *A,B* vom Typ *cmatrix* gilt:

$$\begin{array}{lllllll} \text{a} & <= & \text{b} & \Longleftrightarrow & \text{a[i]} & <= & \text{b[i]} \quad \text{für alle i} \\ \text{A} & <= & \text{B} & \Longleftrightarrow & \text{A[i,j]} & <= & \text{B[i,j]} \quad \text{für alle i, j} \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Werte vom Typ *complex*.

linker Operand \ rechter Operand	integer real	complex	rvector	cvector	rmatrix	cmatrix
monadisch				$+, -$		$+, -$
integer real				$*, *<, *>$		$*, *<, *>$
complex			$*, *<, *>$	$*, *<, *>$	$*, *<, *>$	$*, *<, *>$
rvector		$*, *<, *>,$ $/, /<, />$		$\circ$ $\vee$		
cvector	$*, *<, *>,$ $/, /<, />$	$*, *<, *>,$ $/, /<, />$	$\circ$ $\vee$	$\circ$ $\vee$		
rmatrix		$*, *<, *>,$ $/, /<, />$		$*, *<, *>$		$\circ$ $\vee$
cmatrix	$*, *<, *>,$ $/, /<, />$	$*, *<, *>,$ $/, /<, />$	$*, *<, *>$	$*, *<, *>$	$\circ$ $\vee$	$\circ$ $\vee$

Die Operatoren des Moduls MVC_ARI

$$\circ \in \{+,\ +<,\ +>,\ -,\ -<,\ ->,\ *,\ *<,\ *>\}$$

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

Beispiel 3.5.1:

Für *cv* vom Typ *cvector* und *cM* vom Typ *cmatrix* läßt sich eine Skalierung mit dem Faktor 1/3 durch die Anweisungen

```
cv := cv / 3;
cM := cM / 3;
```

realisieren, was durch Verwendung der Operatoren /< bzw. /> auch mit Rundung nach unten bzw. nach oben durchgeführt werden kann.

Transferfunktionen für komplexe Vektoren

Zur Wandlung zwischen den Typen *rvector* und *cvector* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	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

Transferfunktionen für komplexe Matrizen

Zur Wandlung zwischen den Typen *rmatrix* und *cmatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	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

Überladungen des Zuweisungsoperators

Die komponentenweisen Initialisierungen 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

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	Ergebnistyp	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* · *cM2*
id (cM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *cM*
id (cM1,cM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix *cM1* · *cM2*

cv = *cvector*-Ausdruck, cM, cM1, cM2 = *cmatrix*-Ausdruck

Funktion	Ergebnistyp	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 3.5.2:

Für die komplexen Matrizen *cM, cM1, cM2* vom Typ *cmatrix* ergibt nach Ausführung der Anweisungen

cM1 := conj (transp (cM));
cM2 := herm (cM);

der logische Ausdruck

cM1 = cM2

den Wert *true*.

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 in einem implementierungsabhängigen Standardformat für die komplexen Komponenten.

3.6 Das Modul MVI_ARI Intervall-Matrix/Vektor-Arithmetik

In diesem Modul werden die für das Rechnen mit Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentypen

Die dynamischen Datentypen zur Darstellung von Intervallvektoren und Intervallmatrizen sind entsprechend der Definition

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.

Operatoren

Viele der aus der Mathematik bekannten grundlegenden 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 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:=lbound(a) **to** ubound(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]	‡

‡: maximal genaues Skalarprodukt

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.

Die wie im Falle von Intervallen und komplexen Intervallen mengentheoretisch zu verstehenden Vergleichsoperatoren =, <>, <, <=, >, >= sind auf der Basis von = und <= wie eingangs erwähnt gemäß (VD) realisiert (vgl. Seite 136), wobei für *a,b* vom Typ *ivector* und *A,B* vom Typ *imatrix* gilt:

$$\begin{array}{lllllll} a & <= & b & \Longleftrightarrow & a[i] & <= & b[i] \quad \text{für alle } i \\ A & <= & B & \Longleftrightarrow & A[i,j] & <= & B[i,j] \quad \text{für alle } i, j \end{array}$$

Die Operatoren auf der rechten Seite der Äquivalenzbeziehungen sind dabei die Operatoren für Werte vom Typ *interval*.

Zusätzlich stehen 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* im Abschnitt I_ARI beschrieben wurde.

Beispiel 3.6.1:

Will man das im Abschnitt MV_ARI erwähnte Runge-Kutta-Verfahren intervallmäßig durchführen, so genügt es, im entsprechenden Programm die Variablen *k1, k2, k3, k4, Y* vom Typ *ivector* zu vereinbaren und die Funktion *F* mit Ergebnistyp *ivector* in geeigneter Weise zu definieren. Verwendet man dann das Modul MVI_ARI, so können die Programmteile

```
k1  :=  h * F (x , Y);
k2  :=  h * F (x + h/2, Y + k1/2);
k3  :=  h * F (x + h/2, Y + k2/2);
k4  :=  h * F (x + h , Y + k3);
Y   :=  Y + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
```

unverändert übernommen werden, da dadurch sämtliche Operationen intervallmäßig ausgeführt werden.

linker Operand \ rechter Operand	integer real	interval	rvector	ivector	rmatrix	imatrix
monadisch				$+, -$		$+, -$
integer real				$*$		$*$
interval			$*$	$*$	$*$	$*$
rvector		$*, /$	$+*$	$\diamond$ $=, <>,$ **in** $+*$		
ivector	$*, /$	$*, /$	$\diamond$ $=, <>$ $+*$	$\diamond$ **in**, $\vee, ><$ $+*, **$		
rmatrix		$*, /$		$*$	$+*$	$\diamond$ $=, <>,$ **in** $+*$
imatrix	$*, /$	$*, /$	$*$	$*$	$\diamond$ $=, <>$ $+*$	$\diamond$ **in**, $\vee, ><$ $+*, **$

Die Operatoren des Moduls MVI_ARI

$\diamond \in \{+, -, *\}$

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

Transferfunktionen für Intervallvektoren

Zur Wandlung zwischen den Typen *rvector* und *ivector* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	Bedeutung
intval (rv1,rv2)	*ivector*	Intervallvektor *iv* mit *iv[i] = intval (rv1[i],rv2[i])* (*)
intval (rv)	*ivector*	Punktintervallvektor *iv* mit *iv[i] = intval (rv[i])*
inf (iv)	*rvector*	Vektor *rv* der Untergrenzen mit *rv[i] = inf (iv[i])*
sup (iv)	*rvector*	Vektor *rv* der Obergrenzen mit *rv[i] = sup (iv[i])*

rv, rv1, rv2 = *rvector*-Ausdruck, iv = *ivector*-Ausdruck

(*): Es muß rv1 <= rv2 gelten, sonst tritt ein Laufzeitfehler auf.

Transferfunktionen für Intervallmatrizen

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] = intval (rM1[i,j],rM2[i,j])* (*)
intval (rM)	*imatrix*	Punktintervallmatrix *iM* mit *iM[i,j] = intval (rM[i,j])*
inf (iM)	*rmatrix*	Matrix *rM* der Untergrenzen mit *rM[i,j] = inf (iM[i,j])*
sup (iM)	*rmatrix*	Matrix *rM* der Obergrenzen mit *rM[i,j] = sup (iM[i,j])*

rM, rM1, rM2 = *rmatrix*-Ausdruck, iM = *imatrix*-Ausdruck

(*): Es muß rM1 <= rM2 gelten, sonst tritt ein Laufzeitfehler auf.

Überladungen des Zuweisungsoperators

Die komponentenweisen Initialisierungen von *ivector*- und *imatrix*-Variablen sowie die Wandlungen 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)

i = *interval*-Ausdruck, iv = *ivector*-**Variable**, iM = *imatrix*-**Variable**
r = *real*-Ausdruck, rv = *rvector*-Ausdruck, rM = *rmatrix*-Ausdruck

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. Darüber hinaus wird die komponentenweise definerte 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* · *iM2*
id (iM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *iM*
id (iM1,iM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix *iM1* · *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)*

r = *real*-Ausdruck, iv = *ivector*-Ausdruck,
iM, iM1, iM2 = *imatrix*-Ausdruck

Beispiel 3.6.2:

Eine Intervalleinschließung für den Defekt $D = E - R \cdot A$ aus dem Abschnitt MV_ARI kann unter Verwendung von MVI_ARI mit *A*, *D* und *R* vom Typ *imatrix* durch die Anweisung

D := id (A) − R ∗ A

bzw. unter Verwendung eines Lattenkreuzausdrucks (vgl. Abschnitt 2.4.4) durch die Anweisung

D := ## (id (A) − R ∗ A)

bestimmt werden, wobei die zweite Form die engstmögliche Einschließung für die Defektmatrix D berechnet.

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*-Größen, wobei eine Matrix stets „zeilenweise“ eingelesen wird. Die Ausgabe eines Intervallvektors bzw. einer Intervallmatrix erfolgt wie bei der Eingabe komponentenweise in einem implementierungsabhängigen Standardformat für die Komponentenintervalle.

3.7 Das Modul MVCI_ARI
Komplexe Intervall-Matrix/Vektor-Arithmetik

In diesem Modul werden die für das Rechnen mit komplexen Intervallvektoren und Intervallmatrizen notwendigen Operatoren, Funktionen und Prozeduren bereitgestellt.

Datentypen

Die dynamischen Datentypen zur Darstellung von komplexen Intervallvektoren und komplexen Intervallmatrizen sind entsprechend der Definition

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.

Operatoren

Viele der aus der Mathematik 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:=lbound(a) **to** ubound(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] ‡

‡: maximal genaues Skalarprodukt

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 $<=$ wie eingangs erwähnt gemäß (VD) realisiert (vgl. Seite 136), wobei für *a,b* vom Typ *civector* und *A,B* vom Typ *cimatrix* gilt:

$$\begin{array}{lllllll} \mathrm{a} & <= & \mathrm{b} & \Longleftrightarrow & \mathrm{a[i]} & <= & \mathrm{b[i]} & \text{für alle } \mathrm{i} \\ \mathrm{A} & <= & \mathrm{B} & \Longleftrightarrow & \mathrm{A[i,j]} & <= & \mathrm{B[i,j]} & \text{für alle } \mathrm{i,\, j} \end{array}$$

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 Abschnitt CI_ARI beschrieben wurde.

Die Übersicht über die im Modul MVCI_ARI definierten Operatoren wird, bedingt durch die große Zahl von Operatoren, in zwei Tabellen gegeben. In der ersten Tabelle tauchen keine Matrix-Typen als rechte Operanden auf, die zweite Tabelle zeigt nur Operatoren mit Matrix-Typen als rechte Operanden.

rechter Operand / linker Operand	integer real	complex	interval	cinterval	rvector	cvector	ivector	civector
monadisch								+, −
integer real								*
complex							*	*
interval						*		*
cinterval					*	*	*	*
rvector				*, /		+*		$\diamond$ =, <>, **in** +*
cvector			*, /	*, /	+*	+*	$\diamond$ =, <>, **in** +*	$\diamond$ =, <>, **in** +*
ivector		*, /		*, /		$\diamond$, =, <>, +*		$\diamond$ **in**, $\vee$, ><, +*, **
civector	*, /	*, /	*, /	*, /	$\diamond$, =, <>, +*	$\diamond$, =, <>, +*	$\diamond$ $\vee$, ><, +*, **	$\diamond$ **in**, $\vee$, ><, +*, **
rmatrix				*, /				*
cmatrix			*, /	*, /			*	*
imatrix		*, /		*, /		*		*
cimatrix	*, /	*, /	*, /	*, /	*	*	*	*

Die Operatoren des Moduls MVCI_ARI (Teil 1)

$\diamond \in \{+, -, *\}$

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

linker Operand \ rechter Operand	rmatrix	cmatrix	imatrix	cimatrix
monadisch				$+, -$
integer real				$*$
complex			$*$	$*$
interval		$*$		$*$
cinterval	$*$	$*$	$*$	$*$
rvector				
cvector				
ivector				
civector				
rmatrix		$+*$		$\diamond$ $=, <>,$ **in** $+*$
cmatrix	$+*$	$+*$	$\diamond$ $=, <>,$ **in** $+*$	$\diamond$ $=, <>,$ **in** $+*$
imatrix		$\diamond,$ $=, <>,$ $+*$		$\diamond$ **in**, $\vee, ><,$ $+*, **$
cimatrix	$\diamond,$ $=, <>,$ $+*$	$\diamond,$ $=, <>,$ $+*$	$\diamond$ $\vee, ><,$ $+*, **$	$\diamond$ **in**, $\vee, ><,$ $+*, **$

Die Operatoren des Moduls MVCI_ARI (Teil 2)

$\diamond \in \{+, -, *\}$

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

Transferfunktionen für komplexe Intervallvektoren

Zur Wandlung zwischen den Typen *rvector*, *cvector*, *ivector* und *civector* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	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

Transferfunktionen für komplexe Intervallmatrizen

Zur Wandlung zwischen den Typen *rmatrix*, *cmatrix*, *imatrix* und *cimatrix* werden folgende Transferfunktionen bereitgestellt:

Funktion	Ergebnistyp	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

Überladungen des Zuweisungsoperators

Die komponentenweisen Initialisierungen 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

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	Ergebnistyp	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* · *ciM2*
id (ciM)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen von *ciM*
id (ciM1,ciM2)	*rmatrix*	Einheitsmatrix mit den aktuellen Indexbereichen der Produktmatrix *ciM1* · *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,j])*
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*	Matrix-Epsilonaufblähung *eM* mit *eM[i,j] = blow (ciM[i,j],r)*

n, n1, n2 = *integer*-Ausdruck, r = *real*-Ausdruck
civ = *civector*-Ausdruck, ciM, ciM1, ciM2 = *cimatrix*-Ausdruck

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);

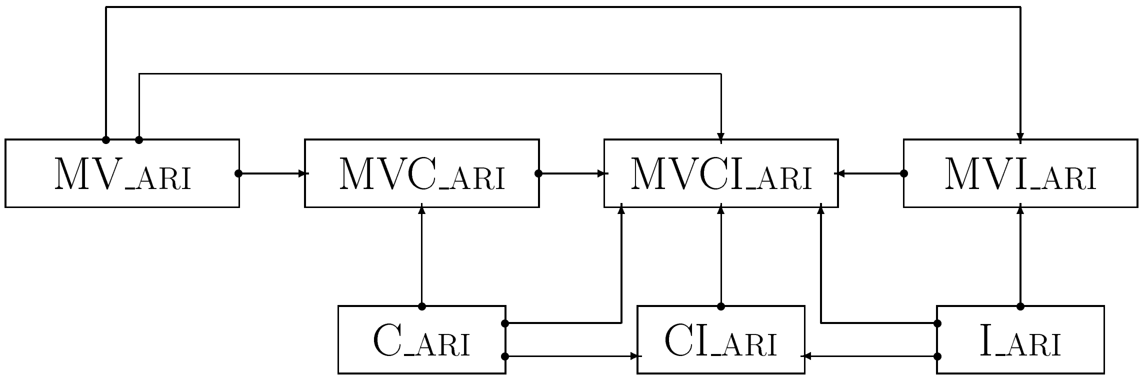
mit optionalem Fileparameter, beliebig vielen Ein-/Ausgabeparametern, jedoch ohne Formatspezifikationen zur Verfügung.

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 in einem implementierungsabhängigen Standardformat für die komplexen Komponentenintervalle.

3.8 Die Hierarchie der Arithmetikmodule

Die wechselseitige Verwendung der Arithmetikmodule untereinander erzeugt eine Art Hierarchie, die in der nachfolgenden Graphik dargestellt ist. Unabhängig davon muß im Anwenderprogramm jedes benutzte Modul in einer **use**-Klausel auftreten, da die hierarchisch niederen Module, durch eine **use**-Klausel ohne Verwendung von **global**, nur „lokal" in die höheren eingebunden sind.

Bei Verwendung des Moduls MVCI_ARI sind z. B. die reinen Intervalloperationen aus I_ARI nicht automatisch bekannt, sondern erst durch die Klausel „**use** I_ARI" ebenfalls verfügbar.



Hierarchie der PASCAL–XSC Arithmetik-Module

hat die Bedeutung „ • *wird benutzt von* ▸"

3.9 Ein vollständiges Beispielprogramm

Im folgenden wird ein vollständiges PASCAL–XSC Programm angegeben, das unter Verwendung des ebenfalls angegebenen Moduls LIN_SOLV eine Einschließung für den Lösungsvektor eines linearen Gleichungssystems berechnet.

Die im Hauptprogramm verwendete Prozedur *main* dient lediglich zum Einlesen der Dimension des Systems bzw. zur Allokierung der dynamischen Variablen. Das eigentliche numerische Verfahren wird durch Aufruf der Prozedur *linear_system_solver* aus dem Modul *lin_solv* gestartet, welche mit beliebiger Dimension aufgerufen werden kann.

Genaueres über das verwendete Iterationsverfahren mit Verifikation des Ergebnisses findet sich z. B. in [34].

Das Hauptprogramm

```
program lin_sys (input,output);

{ Programm zur verifizierten Lösung von Linearen Gleichungssystemen }
{ Die Matrix A und die rechte Seite b des Systems werden eingelesen. }
{ Das Programm liefert entweder eine verfizierte Lösung oder eine    }
{ entsprechende Fehlermeldung.                                       }

use                           { lin_solv : Gleichungssystemlöser          }
  lin_solv, mv_ari, mvi_ari;  { mv_ari   : Matrix/Vektor-Arithmetik        }
                              { mvi_ari  : Matrix/Vektor-Intervallarithmetik }
var
  n : integer;

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

procedure main (n : integer);

{ Die Matrix A und die Vektoren b, x werden durch den Aufruf dieser }
{ Prozedur dynamisch allokiert. Die Matrix A und die rechte Seite b  }
{ werden eingelesen und linear_system_solver wird aufgerufen.       }

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

begin

  writeln('Please enter the matrix A:');
```

```
  read(A);

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

  linear_system_solver(A,b,x,ok);

  if ok then

    begin
      writeln('The given matrix A is non-singular and the solution ');
      writeln('of the linear system is contained in:');
      write(x);
    end

  else

    writeln('No solution found !');

end;   {procedure main}

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

begin

  write('Please enter the dimension n of the linear system: ');
  read(n);
  main(n);

end. {program lin_sys}
```

Das Modul LIN_SOLV

```
module lin_solv;

{ Verifizierte Lösung des linearen Gleichungssystems Ax = b. }

use                          { i_ari   : Intervallarithmetik                 }
  i_ari, mv_ari, mvi_ari;    { mv_ari  : Matrix/Vektor-Arithemtik            }
                             { mvi_ari : Matrix/Vektor-Intervallarithmetik   }
priority
  inflated = *;   { Prioritätsstufe 2 }


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


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

{ Berechnet die sogenannte Epsilon-Aufblähung eines Intervallvektors. }

var
  i : integer;
  x : interval;

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

  infl := a;
end;   {operator inflated}


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

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

{ Berechnung einer Näherungsinverse der (n,n)-Matrix A durch }
{ Anwendung des Gauss-Eliminationsverfahrens.                }

var
  i, j, k, n   : integer;
  factor       : real;
  R, Inv, E : rmatrix[1..ubound(A),1..ubound(A)];
begin
  n := ubound(A);    { Dimension von A }

  E := id(E)    { Einheitsmatrix }
  R := A;

  { Gauss-Eliminationsschritt mit Einheitsvektoren  }
  { als rechte Seite. Eine Division durch R[i,i]=0      }
  { zeigt eine wahrscheinlich singuläre Matrix A an. }
  for i:= 1 to n do
    for j:= (i+1) to n do
    begin
      factor := R[j,i]/R[i,i];
      for k:= i to n do R[j,k] := #*(R[j,k] - factor*R[i,k]);
      E[j] := E[j] - factor*E[i];
    end;    {for j:= ...}

  { Rückwärtssubstitution liefert die Zeilen der Inversen von A. }

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

  approximate_inverse := Inv;
end;    {function approximate_inverse}

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

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

{ Berechnung eines verifizierten Einschließungsvektors für die Lösung    }
{ des linearen Gleichungssystems. Wird keine Einschließung erreicht, so }
{ wird die Iteration nach 10 Schritten abgebrochen und der Parameter    }
{ ok wird auf false gesetzt.                                            }

const
  epsilon   = 0.25; { Konstante für die Epsilon-Aufblähung     }
  max_steps = 10;   { Maximale Anzahl von Iterationsschritten }

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

begin
  R := approximate_inverse(A);

  { R*b ist eine Näherungslösung des linearen Systems und z ist eine }
  { Einschließung dieses Vektors, jedoch nicht der wahren Lösung.     }

  z := ##(R*b);

  { Eine Einschließung von  I - R*A  wird mit maximaler Genauigkeit  }
  { berechnet. Die Einheitsmatrix wird durch den Aufruf id(A) erzeugt. }

  C := ##(id(A) - R*A);

  x := z;    i := 0;
  repeat
      i := i + 1;

      y := x inflated epsilon; { Um eine Einschließung zu erreichen, wird }
                               { der Vektor x leicht aufgebläht.          }

      x := z + C*y;            { Die neue Iterierte wird berechnet }

     ok := x < y;              { Ist x Teilintervall von y ?}

  until ok or (i = max_steps);
end;    {procedure linear_system_solver}

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

end.    {module lin_solv}
```

Kapitel 4

Problemlöseroutinen

In Form einer zusätzlichen Modulbibliothek stehen PASCAL–XSC-Routinen 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 der Lösung im angegebenen Intervall nach. Diese neuartigen Routinen haben also 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 oder fehlinterpretieren, und somit sind diese Methoden auch von Nichtspezialisten einsetzbar.

Im einzelnen decken die PASCAL–XSC Routinen die folgenden Gebiete ab:

- Lineare Gleichungssysteme
  - Vollbesetzte Systeme (*real, complex, interval, cinterval*)
  - Matrixinvertierung (*real, complex, interval, cinterval*)
  - Ausgleichsprobleme (*real, complex, interval, cinterval*)
  - Berechnung der Pseudoinversen (*real, complex, interval, cinterval*)
  - Bandmatrizen (*real*)
  - Spärlich besetzte Matrizen (*real*)
- Polynomauswertung
  - in einer Variablen (*real, complex, interval, cinterval*)
  - in mehreren Variablen (*real*)

- Polynomnullstellen (*real*, *complex*, *interval*, *cinterval*)
- Eigenwerte und Eigenvektoren
  - Symmetrische Matrizen (*real*)
  - Beliebige Matrizen (*real*, *complex*, *interval*, *cinterval*)
- Anfangs- und Randwertprobleme bei gewöhnlichen Differentialgleichungen
  - linear
  - nichtlinear
- Auswertung von arithmetischen Ausdrücken
- Nichtlineare Gleichungssysteme
- Numerische Quadratur
- Integralgleichungen
- Automatische Differentiation

Näheres zu den einzelnen Routinen und Modulen kann der begleitenden Dokumentation zur 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 technischer Einrichtungen 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.
- Hintergrundkontrolle von Realzeitprozessen, welche zeitverschoben vergangene Realzeitprozesse verifizierend nachrechnet und die Genauigkeit und damit die Zuverlässigkeit kontrolliert. Besonderer Anwendungsbereich bei Sicherheitsfragen (Flugzeug- oder Satellitensteuerung, Raumfahrt, hochempfindliche großtechnische Einrichtungen).

Verifizierende Methoden sind einseitige Entscheidungsverfahren, welche auf der Basis der gegebenen Mittel (Rechenzeit, Speicherbedarf, Mantissenlänge usw.) Probleme als lösbar erkennen und auf eine gewünschte Genauigkeit einschließen. Alle anderen Probleme bleiben in diesem Sinn unentscheidbar, d. h. nur mit höherem Aufwand können weitere Probleme lösbar werden. Selbstverständlich werden mathematisch unlösbare Probleme korrekterweise niemals als lösbar (Scheinlösung) ausgegeben.

Kapitel 5

Übungsaufgaben mit Lösungen

In diesem Kapitel wird dem Leser anhand einiger Übungsaufgaben die Möglichkeit gegeben, sich in die Benutzung der Sprache PASCAL–XSC einzuarbeiten und die neuen Sprachelemente in der Entwicklung von vollständigen Programmen anzuwenden. Dazu wird eine Reihe von Aufgaben mit geringem Schwierigkeitsgrad angegeben, die einen großen Teil der wichtigsten Sprachelemente von PASCAL–XSC zur Anwendung bringen. Dabei handelt es sich um

- einführende Aufgaben
- Aufgaben zur Vertiefung der neuen Konzepte von PASCAL–XSC (Operatorkonzept, Funktionen mit allgemeinem Ergebnistyp, dynamische Felder, Modulkonzept usw.)
- einfache Aufgaben zur Behandlung von Genauigkeitsfragen bei arithmetischen Operationen und numerischen Berechnungen (Verwendung des Datentyps *dotprecision*)
- Aufgaben zu diversen Arithmetiken (Intervallarithmetik, komplexe Arithmetik, Matrix/Vektor-Arithmetik etc.)
- Aufgaben zu physikalischen und ingenieurwissenschaftlichen Anwendungen von Programmiersprachen und numerischen Verfahren.

Diese Aufgaben sind größtenteils einer Sammlung entnommen, die im Rahmen der Programmierausbildung an der Universität Karlsruhe im Laufe der letzten Jahre entstanden ist.

Im Anschluß an die Aufgaben finden sich jeweils die Lösungen mit kompletten Programmlistings, außerdem sind in einigen Fällen Ablaufprotokolle zu den jeweiligen Programmen aufgeführt, um die Funktionsweise der angegebenen Implementierung zu demonstrieren. Alle diese Protokolle wurden auf einem ATARI ST Rechner unter Verwendung einer PASCAL–XSC-Version mit 13-stelliger Dezimalarithmetik erzeugt.

Bei Verwendung einer PASCAL–XSC-Implementierung mit Binärarithmetik können infolge der in Kapitel 2 beschriebenen Konvertierungsproblematik Abweichungen in den Ergebnissen auftreten. Aus diesem Grund wurden einige Beispielprogramme zusätzlich auf einem HP 9000 Rechner unter Verwendung einer PASCAL–XSC–Version mit 53-stelliger Binärarithmetik ausgeführt und protokolliert.

Aufgabe 1: Darstellbarkeitstest

In einem PASCAL–XSC Programm soll geprüft werden, ob ein Paar von *integer*-Zahlen z, n ($n \neq 0$) einen Quotienten z/n ergibt, der als *real*-Zahl exakt darstellbar ist.

Hinweis: z/n ist genau dann exakt auf dem Rechner darstellbar, wenn gilt

$$z /< n = z /> n.$$

In einer Schleife sollen beliebig viele solcher Paare eingelesen und geprüft werden. Wenn die genannte Bedingung erfüllt ist, sind z, n und z/n auszugeben. Nach dem Abbruch der Schleife (durch Eingabe mit $n = 0$), soll ausgegeben werden, wieviel Prozent der vorher geprüften Paare einen exakt darstellbaren Quotienten ergaben (Ausgabe auf eine Stelle hinter dem Dezimalpunkt abgerundet).

Lösung:

```
program darstellbar (input,output);

 { Darstellbarkeitstest }

var n, z          : integer;
    exakt, anzahl : integer;
    quot          : real;

begin
  exakt:=  0;
  anzahl:= 0;
  write ('Datenpaar ? ');
  read (z,n);
  while n <> 0 do
  begin
    quot:= z/<n;
    anzahl:= anzahl+1;
    if quot = z/>n then
      begin
        exakt:= exakt+1;
        writeln ('Quotient kann exakt dargestellt werden !');
        writeln (z:1,'/',n:1,' = ',quot);
      end
    else
      writeln ('Quotient kann nicht exakt dargestellt werden !');
    writeln;
    write ('Datenpaar ? ');
    read (z,n);
  end;
```

```
  if anzahl <> 0 then
  begin
    writeln;
    writeln (anzahl, ' Datenpaare wurden eingelesen');
    writeln (exakt,  ' Quotienten konnten exakt dargestellt werden');
    writeln ('Dies sind ', exakt/anzahl*100:5:1:-1,'%');
  end;
end.
```

Aufgabe 2: Berechnen der Exponentialreihe

Die Funktion e^x wird durch eine Teilsumme ihrer *Taylor*-Reihe angenähert:

$$S_n = \sum_{i=0}^{n} a_i \quad \text{mit} \quad a_i = \frac{x^i}{i!} \quad \text{und} \quad i! = \begin{cases} 1 & \text{für } i = 0 \\ 1 * 2 * \cdots * i & \text{für } i > 0 \end{cases} .$$

Die Teilsumme läßt sich nach folgendem Algorithmus berechnen:

$$\begin{array}{rl} \text{Start:} & S_1 = 1;\;\; a_1 = x \\ \text{Rekursion:} & S_i = S_{i-1} + a_{i-1};\quad a_i = a_{i-1}\dfrac{x}{i};\quad i = 2, \ldots, n \end{array} .$$

Schreiben Sie ein PASCAL–XSC Programm mit $n = 100$, das x einliest und S_n mit drei verschiedenen Rundungskontrollen berechnet:

- nach unten gerichtet
- zur nächsten Gleitkommazahl
- nach oben gerichtet.

Die Berechnung der Summe werde vor dem n-ten Summanden abgebrochen, wenn für die mit der nach unten gerichteten Rundung berechnete Summe S_i gilt:

$|a_i| < eps * |S_i|$ mit $eps = 10^{-12}$.

Am Ende soll das zuletzt berechnete Tripel für S_i sowie der Wert von i und $exp(x)$ ausgegeben werden.

Hinweis: Verwenden Sie beim Test Ihres Programms auch negative Werte (< -50) für x, da an diesen sehr deutlich die bei der Summation auftretenden Rundungsfehler demonstriert werden.

Lösung:

```
program expo (input,output);

  { Berechnen der Exponentialreihe }

const eps = 1e-12;
      n   = 100;

var  Sunten, Snaechste, Soben : real;
     aunten, anaechste, aoben : real;
     i                        : integer;
     x, hilf                  : real;

begin
  write ('Geben Sie ein Argument ein: ');
```

```
  read (x);
  writeln ('Berechnen der Exponentialreihe :');
  writeln ('Schritt      Summand                 Summe');
  aunten:= x; anaechste:= x; aoben:= x;
  Sunten:= 1; Snaechste:= 1; Soben:= 1;
  i:= 1;
  repeat
    i:=i+1;
    Sunten   := Sunten    +< aunten;
    Snaechste:= Snaechste +  anaechste;
    Soben    := Soben     +> aoben;
    anaechste:= anaechste *  x /  i;
    if x >= 0 then
      begin
        aunten   := aunten *< x /< i;
        aoben    := aoben  *> x /> i;
      end
    else
      begin
        hilf     := aunten;
        aunten   := aoben *< x /< i;
        aoben    := hilf  *> x /> i;
      end;
    writeln (i  :7,' ', aunten,   ' ', Sunten);
    writeln (' ':7,' ', anaechste,' ', Snaechste);
    writeln (' ':7,' ', aoben,    ' ', Soben);
  until (i >= n) or (abs(aunten)<eps*abs(Sunten));
  writeln('Exakter Wert der Exp-Funktion : ',exp(x));
end.
```

Aufgabe 3: Rundungsfehlereinflüsse

Schreiben Sie ein PASCAL–XSC Programm zur Demonstration der verschiedenen Rundungsfehlereinflüsse bei der Berechnung des Ausdrucks

$$z = x^4 - 4y^4 - 4y^2$$

für verschiedene Werte von x und y. Das Programm soll die *real*-Werte x und y einlesen und z nach folgenden Methoden berechnen:

1) $z = x \cdot x \cdot x \cdot x - 4 \cdot y \cdot y \cdot y \cdot y - 4 \cdot y \cdot y$ unter Verwendung

   a) der Gleitkommaoperatoren `*` und `-`

   b) der gerichteten Operatoren `*<`, `*>` und `-<`, so daß der Ausdruck nach unten gerundet berechnet wird

   c) der gerichteten Operatoren `*<`, `*>` und `->`, so daß der Ausdruck nach oben gerundet berechnet wird

2) $z = x^2 \cdot x^2 - 4 \cdot y^2 \cdot y^2 - 4 \cdot y^2$ unter Verwendung der Standardfunktion `SQR` und den Operatoren `*` und `-`

3) $z = (x^2)^2 - (2 \cdot y^2)^2 - (2 \cdot y)^2$ ebenfalls mit `SQR`, `*` und `-`

4) $z = (x^2)^2 - (2 \cdot y)^2 \cdot (y^2 + 1)$ ebenfalls mit `SQR`, `*` und `-`

5) $z = \# * (a \cdot a - b \cdot b - c \cdot c)$ mit $a = x^2$, $b = 2 \cdot y^2$ und $c = 2 \cdot y$.

Die sieben berechneten Werte sollen mit begleitendem Kommentar zur jeweiligen Operation ausgegeben werden. Testen Sie Ihr Programm auch mit den Werten

$$x = 665857.0 \quad \text{und} \quad y = 470832.0.$$

Der korrekte Wert für z ist in diesem Fall 1.

Lösung:

```
program Rundung (input,output);

var x,y,z: real;
    a,b,c: real;

begin
  writeln('Rundungsfehlereinfluesse');
  write('x = '); read(x);
  write('y = '); read(y);
  writeln;
  writeln('Berechnung des Ausdrucks z = x^4 - 4y^4 - 4y^2');
  writeln;
  z:= x*x*x*x-4*y*y*y*y-4*y*y;
```

```
  writeln('Ber.: x*x*x*x-4*y*y*y*y-4*y*y                      = ',z);
  z:= (x*<x)*<(x*<x) -< 4*>(y*>y)*>(y*>y) -< 4*>(y*>y);
  writeln('Ber.: (x*<x)*<(x*<x)-<4*>(y*>y)*>(y*>y)-<4*>(y*>y) = ',z);
  z:= (x*>x)*>(x*>x) -> 4*<(y*<y)*<(y*<y) -> 4*<(y*<y);
  writeln('Ber.: (x*>x)*>(x*>x)->4*<(y*<y)*<(y*<y)->4*<(y*<y) = ',z);
  z:= sqr(x)*sqr(x) - 4*sqr(y)*sqr(y) - 4*sqr(y);
  writeln('Ber.: x^2*x^2-4*y^2*y^2-4*y^2                      = ',z);
  z:= sqr(sqr(x))-sqr(2*sqr(y)) - sqr(2*y);
  writeln('Ber.: (x^2)^2-(2*y^2)^2-(2*y)^2                    = ',z);
  z:= sqr(sqr(x))-sqr(2*y) * (sqr(y)+1);
  writeln('Ber.: (x^2)^2-(2*y)^2*(y^2+1)                      = ',z);
  a:=sqr(x);
  b:=2*sqr(y);
  c:=2*y;
  z:=#*(a*a-b*b-c*c);
  writeln('Ber.: #*(x^2*x^2-(2*y^2)*(2*y^2)-(2*y)*(2*y))      = ',z);
end.
```

Ablaufprotokoll:

```
Rundungsfehlereinfluesse
x = 665857.0
y = 470832.0

Berechnung des Ausdrucks z = x^4-4y^4-4y^2
```

a) mit 13-stelliger Dezimalarithmetik

```
Ber.: x*x*x*x-4*y*y*y*y-4*y*y                      =  1.326891110400E+10
Ber.: (x*<x)*<(x*<x)-<4*>(y*>y)*>(y*>y)-<4*>(y*>y) = -8.673108889600E+10
Ber.: (x*>x)*>(x*>x)->4*<(y*<y)*<(y*<y)->4*<(y*<y) =  1.132689111040E+11
Ber.: x^2*x^2-4*y^2*y^2-4*y^2                      =  1.326891110400E+10
Ber.: (x^2)^2-(2*y^2)^2-(2*y)^2                    =  1.326891110400E+10
Ber.: (x^2)^2-(2*y)^2*(y^2+1)                      =  0.000000000000E+00
Ber.: #*(x^2*x^2-(2*y^2)*(2*y^2)-(2*y)*(2*y))      =  1.000000000000E+00
```

b) mit 53-stelliger Binärarithmetik

```
Ber.: x*x*x*x-4*y*y*y*y-4*y*y                      =  1.1885568000000E+007
Ber.: (x*<x)*<(x*<x)-<4*>(y*>y)*>(y*>y)-<4*>(y*>y) = -5.5223296000000E+007
Ber.: (x*>x)*>(x*>x)->4*<(y*<y)*<(y*<y)->4*<(y*<y) =  1.1885568000000E+007
Ber.: x^2*x^2-4*y^2*y^2-4*y^2                      =  1.1885568000000E+007
Ber.: (x^2)^2-(2*y^2)^2-(2*y)^2                    =  1.1885568000000E+007
Ber.: (x^2)^2-(2*y)^2*(y^2+1)                      =  0.0000000000000E+000
Ber.: #*(x^2*x^2-(2*y^2)*(2*y^2)-(2*y)*(2*y))      =  1.0000000000000E+000
```

Bemerkung: Der #-Ausdruck liefert in beiden Fällen das exakte Ergebnis. Voraussetzung dafür ist, dem Konzept der #-Ausdrücke entsprechend, daß alle Eingangsdaten (hier a, b, c) exakt sind. Diese Voraussetzung ist bei den gegebenen Eingabedaten x, y für beide Gleitkommaformate erfüllt.

Aufgabe 4: Skalarprodukt

In einem PASCAL–XSC Programm soll der maximal genau berechnete Wert eines Skalarprodukts

$$x \cdot y = \sum_{i=1}^{n} x_i \cdot y_i$$

zweier reeller Vektoren $x, y \in \mathbb{R}^n$ mit dem durch herkömmliche Rechnung erzeugten Wert verglichen werden.

Implementieren Sie dazu eine Funktion `SKALP`, die die Skalarproduktbildung auf herkömmliche Art durchführt, und eine Funktion `MAX_GEN_SKALP`, die das Skalarporodukt durch Aufsummierung der Produkte $x_i \cdot y_i$ in einer Variablen vom Typ *dotprecision* und einmaliger abschließender Rundung zu einem *real*-Wert berechnet.

Die Vektoren x und y sollen im Hauptprogramm eingelesen werden, die mit `SKALP` und `MAX_GEN_SKALP` berechneten Werte sollen mit Kommentaren ausgegeben werden. Wählen Sie $n = 5$ für die Vereinbarung der Vektor-Typen. Testen Sie das Programm mit den Vektoren

$$x = \begin{pmatrix} 2.718281828 \\ -3.141592654 \\ 1.414213562 \\ 0.5772156649 \\ 0.3010299957 \end{pmatrix}, \qquad y = \begin{pmatrix} 1486.2497 \\ 878366.9879 \\ -22.37492 \\ 4773714.647 \\ 0.000185049 \end{pmatrix}.$$

Lösung:

```
program Skalarprodukt (input,output);

const n = 5;
type  vektor = array [1..n] of real;
var   x, y : vektor;
      i    : integer;

function SKALP (x, y : vektor) : real;
  var s : real;
      i : integer;
  begin
    s:= 0;
    for i:=1 to n do  s:= s + x[i]*y[i];
    SKALP:= s;
  end;

function MAX_GEN_SKALP (x, y : vektor) : real;
  var d : dotprecision;
      i : integer;
```

```
  begin
    d:= #(0);
    for i:=1 to n do  d:= #(d + x[i]*y[i]);
    MAX_GEN_SKALP:= #*(d);
  end;

begin
  writeln('1. Vektor (mit ',n:1,' Komponenten) eingeben:');
  for i:=1 to n do  read(x[i]);
  writeln('2. Vektor (mit ',n:1,' Komponenten) eingeben:');
  for i:=1 to n do  read(y[i]);
  writeln;
  writeln('Skalarprodukt herkoemmlich berechnet: ',SKALP(x,y));
  writeln('Skalarprodukt mit Dotprecision      : ',MAX_GEN_SKALP(x,y));
end.
```

Ablaufprotokoll:

```
1. Vektor (mit 5 Komponenten) eingeben:
2.718281828  -3.141592654  1.414213562  0.5772156649  0.3010299957
2. Vektor (mit 5 Komponenten) eingeben:
1486.2497  878366.9879  -22.37492  4773714.647  0.000185049
```

a) mit 13-stelliger Dezimalarithmetik

```
Skalarprodukt herkoemmlich berechnet: -2.947003257100E-07
Skalarprodukt mit Dotprecision      : -1.006571070000E-11
```

b) mit 53-stelliger Binärarithmetik

```
Skalarprodukt herkoemmlich berechnet:  1.02518813682967E-010
Skalarprodukt mit Dotprecision      : -9.55468933463301E-011
```

Bemerkung: Der zunächst verblüffende Unterschied der mit *dotprecision* berechneten Werte entsteht durch die Konvertierungsfehler, die im Falle der Binärarithmetik bei der Eingabe unvermeidbar entstehen. Bezogen auf die exakten dezimalen Eingabedaten ist also das unter a) aufgeführte *dotprecision*-Ergebnis von maximaler Genauigkeit.

Aufgabe 5: Boothroyd/Dekker-Matrizen

Die (ganzzahligen) Elemente einer n-dimensionalen Boothroyd/Dekker-Matrix $D = (d_{ij})$ sind gegeben durch

$$d_{ij} = \binom{n+i-1}{i-1} \cdot \binom{n-1}{n-j} \cdot \frac{n}{i+j-1} \quad .$$

Schreiben Sie ein PASCAL–XSC Programm, das unter Verwendung eines Operators `CHOOSE` für die (ganzzahlige) Berechnung des Binomialkoeffizienten $\binom{m}{k}$ eine n-dimensionale Boothroyd/Dekker-Matrix berechnet und zeilenweise ausgibt. Der Wert n (≤ 10) ist einzulesen.

Hinweis: Berechnen Sie $\binom{m}{k}$ nach folgendem Algorithmus:

$$c_0 := 1; \quad c_i := c_{i-1} * \frac{m-i+1}{i}; \quad i = 1, \ldots, k$$

$$\binom{m}{k} := c_k;$$

Beachten Sie, daß die *integer*-Division **div** zu verwenden ist.

Lösung:

```
program BDM (input,output);

 { Boothroyd/Dekker-Matrizen }

var i, j, n, d : integer;

priority CHOOSE = *;

operator CHOOSE (m, k: integer) CHOOSEERG : integer;
var i, c: integer;
begin
  c:= 1;
  for i:=1 to k do
    c:= (c*(m-i+1)) div i;
  CHOOSEERG:= c;
end;

begin
  writeln('Boothroyd/Dekker-Matrix');
  write('Dimension (<=10): ');
  read (n);
  for i:=1 to n do
```

```
  begin
    for j:=1 to n do
    begin
      d:=(((n+i-1) CHOOSE (i-1))*((n-1) CHOOSE (n-j))*n) div (i+j-1);
      write (d:8);
    end;
    writeln;
  end;
end.
```

Ablaufprotokoll:

```
Boothroyd/Dekker-Matrix
Dimension (<=10): 8

     8      28      56      70      56      28       8       1
    36     168     378     504     420     216      63       8
   120     630    1512    2100    1800     945     280      36
   330    1848    4620    6600    5775    3080     924     120
   792    4620   11880   17325   15400    8316    2520     330
  1716   10296   27027   40040   36036   19656    6006     792
  3432   21021   56056   84084   76440   42042   12936    1716
  6435   40040  108108  163800  150150   83160   25740    3432
```

Aufgabe 6: Komplexe Funktionen

Schreiben Sie ein PASCAL–XSC Programm, das für 20 einzugebende komplexe Zahlen $z = x + iy$ die Werte e^z, $\cos z$, $\sin z$, $\cosh z$, und $\sinh z$ berechnet und als Tabelle ausgibt.

Verwenden Sie den Standardtyp *complex* und definieren Sie sich einen monadischen Operator `IMAL`, der die Multiplikation einer komplexen Zahl mit der imaginären Einheit i realisiert, einen monadischen Operator `-` für komplexe Zahlen, zwei Operatoren `+` und `-` für die Addition und Subtraktion zweier komplexer Zahlen, einen Operator `*` für die Multiplikation einer *real*-Zahl mit einer komplexen Zahl, entsprechende Funktionen `exp`, `cos`, `sin`, `cosh`, `sinh` unter Verwendung der Standardfunktionen `sin`, `cos` und `exp` für *real*-Größen sowie Prozeduren für die Ein- und Ausgabe.

Hinweise: $(u, z \in \mathbb{C}; \quad x, y, v, w \in \mathbb{R})$

$u = iz$ mit $u = v + iw$ ergibt sich durch: $v = -y$ und $w = x$.

Division durch i ist durch Multiplikation mit $-i$ zu ersetzen.

real-Divisionen sind durch Multiplikationen zu ersetzen.

$$e^z = e^x \cos y + ie^x \sin y$$

$$\cos z = \frac{e^{iz} + e^{-iz}}{2} \qquad \cosh z = \frac{e^z + e^{-z}}{2}$$

$$\sin z = \frac{e^{iz} - e^{-iz}}{2i} \qquad \sinh z = \frac{e^z - e^{-z}}{2}$$

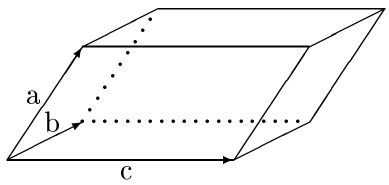
Lösung:

```
program komplfunk (input,output);

 { Komplexe Funktionen }

var z : complex;
    c : array [1..20] of complex;
    i : integer;

priority imal = ^;
operator imal (z : complex) mali : complex;
begin
  mali.re:= -z.im;
  mali.im:=  z.re;
end;
```

```
operator + (a, b: complex) cplus : complex;
begin
  cplus.re:= a.re + b.re;
  cplus.im:= a.im + b.im;
end;

operator - (a, b: complex) cminus : complex;
begin
  cminus.re:= a.re - b.re;
  cminus.im:= a.im - b.im;
end;

operator - (a: complex) cvorz : complex;
begin
  cvorz.re:= -a.re;
  cvorz.im:= -a.im;
end;

operator * (r: real; z: complex) mulrc : complex;
begin
  mulrc.re:= r * z.re;
  mulrc.im:= r * z.im
end;

function exp (z: complex) : complex;
begin
  exp.re:= exp (z.re) * cos(z.im);
  exp.im:= exp (z.re) * sin(z.im);
end;

function cos (z: complex) : complex;
begin
  cos:= 0.5 * (exp (imal z) + exp (- imal z))
end;

function sin (z: complex) : complex;
begin
  sin:= 0.5 * - imal (exp (imal z) - exp (-imal z))
end;

function cosh (z: complex) : complex;
begin
  cosh:= 0.5 * (exp (z) + exp (-z))
end;
```

```
function sinh (z: complex) : complex;
begin
  sinh:= 0.5 * (exp (z) - exp (-z));
end;

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

procedure read (var f: text; var c: complex);
begin
  read (f, c.re, c.im);
end;

begin
  writeln('Komplexe Funktionen');
  for i:=1 to 20 do
  begin
    write (i:2,'. komplexe Zahl eingeben: ');
    read  (c[i]);
  end;
  writeln;
  writeln(' z ':10,' exp(z)':20,' cos(z) ':20,' sin(z) ':20);
  for i:=1 to 20 do
  begin
    z:= c[i];
    write   (z:8);
    write   (exp(z):8);
    write   (cos(z):8);
    writeln (sin(z):8);
  end;
  writeln;
  writeln(' z ':10,' cosh(z)':20,' sinh(z) ':20);
  for i:=1 to 20 do
  begin
    z:= c[i];
    write   (z:8);
    write   (cosh(z):8);
    writeln (sinh(z):8);
  end;
end.
```

Aufgabe 7: Oberfläche eines Parallelflachs



Es soll die Oberfläche eines Parallelflachs im dreidimensionalen Anschauungsraum berechnet werden. Dazu werden folgende Begriffe verwendet:

- Es bedeuten $a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$, $b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$ und $c = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$ Vektoren im Anschauungsraum.
- Das Skalarprodukt $a \cdot b$ (Punktprodukt) zweier Vektoren a und b ist definiert als
$$a \cdot b = \sum_{i=1}^{3} a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3.$$
- Die Länge L eines Vektors a wird berechnet als $L(a) := \sqrt{a \cdot a}$.
- Das Vektorprodukt $a \times b$ (Kreuzprodukt) zweier Vektoren ergibt wieder einen Vektor, der definiert ist durch
$$a \times b := \begin{pmatrix} a_2 \cdot b_3 - a_3 \cdot b_2 \\ a_3 \cdot b_1 - a_1 \cdot b_3 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix}.$$
- Der Flächeninhalt eines Parallelogramms, das durch a und b aufgespannt wird, ergibt sich zu $F(a,b) := L(a \times b)$.
- Die Oberfläche eines Parallelflachs, aufgespannt durch die Vektoren a, b und c, ist gegeben durch
$$OB(a,b,c) := 2 \cdot (F(a,b) + F(b,c) + F(c,a)).$$

Schreiben Sie ein PASCAL–XSC Programm, das die folgenden Teile enthält:

a) eine Typvereinbarung `VEKTOR`, worin ein Vektor als Feld der Länge 3 festgelegt wird (Komponententyp *real*),

b) einen Operator $*$ für das Skalarprodukt zweier Vektoren,

c) eine Funktion `L` für die Länge eines Vektors,

d) einen Operator `KREUZ` für das Vektorprodukt zweier Vektoren,

e) eine Funktion `F` für den Flächeninhalt eines Parallelogramms,

f) eine Funktion `OB` für die Oberfläche eines Parallelflachs,

g) ein Hauptprogramm, das von einem File mit Komponententyp `VEKTOR` bis zum Ende des Files jeweils drei Vektoren einliest, die Oberfläche des zugehörigen Parallelflachs berechnet und ausdruckt.

Hinweis: Das Eingabefile enthält eine durch 3 teilbare Anzahl von Vektoren.

Die Vektorprodukte b) und d) sollen mit Hilfe von *dotprecision*-Ausdrücken maximal genau ausgeführt werden.

Lösung:

```
program paraflach (Datei,input,output);

 { Oberflaeche eines Parallelflachs }

type
  VEKTOR = array [1..3] of real;

operator * (a, b: VEKTOR) skalp : real;
var
  i : integer;
begin
  skalp:= #* (for i:=1 to 3 sum (a[i]*b[i]));
end;

function L (a: VEKTOR) : real;
begin
  L:= sqrt (a*a);
end;

priority KREUZ = *;

operator KREUZ (a, b: VEKTOR) kprod : VEKTOR;
begin
  kprod[1]:=#*(a[2]*b[3]-a[3]*b[2]);
  kprod[2]:=#*(a[3]*b[1]-a[1]*b[3]);
  kprod[3]:=#*(a[1]*b[2]-a[2]*b[1]);
end;

function F (a, b : VEKTOR) : real;
begin
  F:= L (a KREUZ b);
end;
```

```
function OB (a, b, c : VEKTOR) : real;
begin
  OB:= 2 * (F(a,b) + F(b,c) + F(c,a));
end;

var
  Datei   : file of VEKTOR;
  a, b, c : VEKTOR;

begin {HP}
  reset (Datei);
  repeat
    read (Datei, a);
    read (Datei, b);
    read (Datei, c);
    writeln ('a : ',a[1],'  ',a[2],'  ',a[3]);
    writeln ('b : ',b[1],'  ',b[2],'  ',b[3]);
    writeln ('c : ',c[1],'  ',c[2],'  ',c[3]);
    write   ('Oberflaeche des Parallelflachs : ');
    writeln (OB(a,b,c));
  until eof(Datei)
end.
```

Aufgabe 8: Parallelität und Geradenschnitt

In der euklidischen Ebene sind zwei Geraden der Form $a_1x+b_1y = c_1$ und $a_2x+b_2y = c_2$ auf Parallelität zu untersuchen. Falls die Geraden nicht parallel sind, soll ihr Schnittpunkt S bestimmt werden. Schreiben Sie ein PASCAL–XSC-Programm, das folgende Teile enthält:

a) eine Typvereinbarung *gerade*, worin eine Gerade als 3-komponentiges Feld mit Komponententyp *real* festgelegt wird,

b) eine Typvereinbarung *punkt*, worin ein Punkt als *record* mit den Komponenten x und y vom Typ *real* festgelegt wird,

c) eine reelle Funktion *det* mit den Argumenten A, B, C, D vom Typ *real* für die Berechnung der Determinante

$$\det = \begin{vmatrix} A & B \\ C & D \end{vmatrix} = A * D - B * C$$

unter Verwendung eines *dotprecision*-Ausdrucks mit Rundung zur betragsmäßig größeren Zahl (Rundung nach außen),

d) einen logischen Operator *parallel_zu*, der für zwei Geraden g_1 und g_2 genau dann den Wert TRUE liefert, wenn gilt $\det(a_1, b_1, a_2, b_2) = 0$,

e) einen Operator $**$ für den Schnitt zweier Variablen g_1, g_2 vom Typ *gerade* mit Ergebnistyp *punkt*, der unter Verwendung von c) die Koordinaten des Schnittpunkts $s = (x_s, y_s)$ der Geraden g_1 (gegeben durch a_1, b_1, c_1) und g_2 (gegeben durch a_2, b_2, c_2) berechnet gemäß

$$x_s = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, \qquad y_s = \frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}.$$

f) ein Hauptprogramm, das in einer Schleife jeweils die Parameter zweier Geraden einliest, mit dem Operator *parallel_zu* prüft, ob die beiden Geraden parallel sind, falls ja die Meldung „Die Geraden sind parallel“ ausgibt und falls nein mit Teil e) ihren Schnittpunkt berechnet und ausgibt, und zwar solange, bis $a_1 = b_1 = 0$ oder $a_2 = b_2 = 0$ erfüllt ist.

Lösung:

```
program parallel (input,output);

type
  komp   = (a,b,c);
  gerade = array [a..c] of real;
  punkt  = record
             x, y : real;
           end;
var
  g1, g2 : gerade;
  s      : punkt;

function det (A, B, C, D: real): real;
  var
    dp : dotprecision;
  begin
    dp:= #(A*D - B*C);
    if sign(dp) = 0 then
      det:= 0
    else if sign(dp) < 0 then
      det:= #< (dp)
    else
      det:= #> (dp);
  end;

priority parallel_zu = =;
operator parallel_zu (g1, g2: gerade) par: boolean;
  begin
    par:= (det(g1[a],g1[b],g2[a],g2[b]) = 0);
  end;

operator ** (g1, g2: gerade) sch: punkt;
  begin
    sch.x:= det(g1[c],g1[b],g2[c],g2[b]) / det(g1[a],g1[b],g2[a],g2[b]);
    sch.y:= det(g1[a],g1[c],g2[a],g2[c]) / det(g1[a],g1[b],g2[a],g2[b])
  end;

begin {Hauptprogramm}
  repeat
    writeln ('Werte fuer a1, b1 und c1 eingeben: ');
    read    (g1[a], g1[b], g1[c]);
    writeln;
    writeln ('Werte fuer a2, b2 und c2 eingeben: ');
    read    (g2[a], g2[b], g2[c]);
    writeln;
```

```
    if g1 parallel_zu g2 then
      writeln ('Die beiden Geraden sind parallel !')
    else
      begin
        s:= g1 ** g2;
        writeln ('Die beiden Geraden schneiden sich im Punkt');
        writeln ('(xs,ys) = (',s.x,',',s.y,')');
      end;
    writeln; writeln;
  until ((g1[a]=0) and (g1[b]=0)) or ((g2[a]=0) and (g2[b]=0));
end.
```

Aufgabe 9: Transponierte einer Matrix, Symmetrie

Eine $n \times n$-Matrix $A = (a_{ij})$ heißt *symmetrisch*, falls für alle $i, j \in \{1, \ldots, n\}$ gilt: $a_{ij} = a_{ji}$. Die *Transponierte* $T = (t_{ij}) = A^T$ einer Matrix A ist definiert durch: $t_{ij} = a_{ji}$ für alle $i, j \in \{1, \ldots, n\}$.

Schreiben Sie ein Programm, das

- einen dynamischen Typ `MATRIX` für *ganzzahlige* Matrizen definiert,
- einen Operator `=` für zwei Matrizen vom Typ `MATRIX` definiert,
- einen monadischen Operator `transp` vereinbart, der die Transponierte einer Matrix liefert,
- eine logische Funktion `symm` deklariert, die genau dann den Wert `TRUE` liefert, wenn ihr Argument (vom Typ `MATRIX`) symmetrisch ist,
- eine Prozedur `MAIN` enthält, die abhängig vom Parameter n zwei quadratische Matrizen A und B vereinbart, A und B einliest, feststellt, ob A und B symmetrisch sind bzw. ob eventuell $A^T = B$ gilt, und entsprechende Informationen ausgibt,
- im Hauptprogramm den Wert n einliest und die Prozedur `MAIN` aufruft.

Hinweis: Beachten Sie, daß die Funktion `symm` sehr einfach mit Hilfe der Operatoren `=` und `transp` formuliert werden kann.

Lösung:

```
program transpsym (input,output);

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

operator = (a, b: matrix) mmgleich: boolean;

var l   : boolean;
    i, j: integer;

begin
  l:= true;
  for i:= lbound(a,1) to ubound (a,1) do
    for j:= lbound(a,2) to ubound(a,2) do
      if a[i,j] <> b[i,j] then
        l:= false;
  mmgleich:= l;
end;

priority transp = ^;
```

```
operator transp (a: matrix) restr:
      matrix[lbound(a,2)..ubound(a,2),lbound(a,1)..ubound(a,1)];

var i, j :integer;

begin
  for i:=lbound(a,2) to ubound(a,2) do
    for j:=lbound(a,1) to ubound(a,1) do
      restr[i,j]:= a[j,i];
end;

function symm (a: matrix): boolean;
begin
  symm:=  a = transp a;
end;

procedure MAIN (n: integer);
var i, j      : integer;
    A, B, At : matrix[1..n,1..n];
begin
  writeln('Matrixelemente von A eingeben:');
  for i:=1 to n do
  begin
    write (i:3,'-te Zeile : ');
    for j:=1 to n do read (A[i,j]);
  end;
  writeln(' Matrixelemente von B eingeben:');
  for i:=1 to n do
  begin
    write(i:3,'-te Zeile : ');
    for j:=1 to n do read (B[i,j]);
  end;
  writeln ('Transponierte von A:');
  At:= transp A;
  for i:=1 to n do
  begin
    for j:=1 to n do write (At[i,j]:5:1);
    writeln;
  end;
  if symm(A) then
    writeln('A ist symmetrisch ');
  if symm(B) then
    writeln('B ist symmetrisch ');
  if transp A = B then
    writeln('Transp (A) ist gleich B ');
end;
```

```
var
  n : integer;

begin
  writeln('Transponierte einer Matrix, Symmetrie');
  write('Dimension n eingeben: ');
  read (n);
  MAIN (n);
end.
```

Aufgabe 10: Streckenplan

Für eine Eisenbahnlinie soll ein Streckenplan erstellt werden. Ausgehend vom Startpunkt `ORT_0` soll ein Zielbahnhof `ORT_9` über 8 Zwischenstationen `ORT_1`,...,`ORT_8` angefahren werden, wobei an jeder dieser Stationen zwei Minuten Aufenthalt vorzusehen sind.

Schreiben Sie ein Programm, das die Abfahrtszeit in `ORT_0` und die Entfernungen zwischen den Stationen `ORT_i` und `ORT_i+1` für $i = 0, \ldots, 8$ einliest. Auf der Basis einer Durchschnittsgeschwindigkeit von 115 km/h sollen anschließend die Ankunfts- und Abfahrtszeiten in den Stationen berechnet und eine Streckentabelle ausgegeben werden.

Vereinbaren Sie dazu

a) einen Typ `UHRZEIT` als *record* mit den Komponenten `STD` und `MIN`,

b) eine Funktion `FAHRZEIT`, die anhand des Streckenabschnitts die Fahrzeit berechnet,

c) einen Operator für die Addition der Fahr- und Aufenthaltszeiten zur aktuellen Uhrzeit,

d) ein Hauptprogramm, das die erforderlichen Daten einliest, die Fahr- und Aufenthaltszeiten berechnet und in einer Tabelle jeweils Ort, Ankunftszeit, Abfahrtszeit und Entfernung zur nächsten Station ausgibt.

Hinweis: Zur Berechnung der Fahrzeiten innerhalb der Funktion `FAHRZEIT` und zur Realisierung des Operators ist es sinnvoll, in Sekunden zu rechnen und diese dann in eine auf volle Minuten gerundete Uhrzeit umzuwandeln.

Lösung:

```
program plan (input,output);
type
  UHRZEIT = record
             STD: 0..23;
             MIN: 0..59;
            end;

var
  i                    : integer;
  strecke              : array [1..9] of real;
  akt_zeit, abfahrt,
  ankunft, aufenthalt  : UHRZEIT;

function FAHRZEIT (str_i: integer) : UHRZEIT;
var
  hours : real;
```

```
  help  : UHRZEIT;

begin
  hours    := strecke[str_i]/115;
  help.STD := trunc(hours);
  hours    := hours - help.STD;
  help.MIN := trunc(hours*60);
  FAHRZEIT := help;
end;

operator + (a, b: UHRZEIT) summe: UHRZEIT;
var hilf : 0..119;
begin
  hilf      := a.MIN + b.MIN;
  summe.MIN := hilf mod 60;
  hilf      := a.STD + b.STD + hilf div 60;
  summe.STD := hilf mod 24
end;

begin
 aufenthalt.STD := 0;
 aufenthalt.MIN := 2;
 write('Bitte Abfahrtszeit eingeben (hh mm): ');
 read (abfahrt.STD, abfahrt.MIN);
 for i:=1 to 9 do
 begin
    write ('Bitte Entfernung zwischen Bahnhof ',
            i-1:1,' und ',i:1,' eingeben (in km) : ');
    read (strecke[i]);
 end;
 writeln;
 writeln('Bahnhof      Ankunft    Abfahrt   Entf. zur naechsten Station');
 writeln('============================================================');
 akt_zeit:= abfahrt;
 writeln(' ORT_','0        --:--        ',
         akt_zeit.STD:2,':',akt_zeit.MIN:2,
         '          ',strecke[1]:10:2,' km');
 for i:=1 to 8 do
 begin
   ankunft  := akt_zeit + FAHRZEIT(i);
   akt_zeit := ankunft + aufenthalt;
   writeln(' ORT_',i:1,'        ',
           ankunft.STD:2,':',ankunft.MIN:2,'       ',
           akt_zeit.STD:2,':',akt_zeit.MIN:2,
           '          ',strecke[i+1]:10:2,' km');
 end;
 ankunft  := akt_zeit + FAHRZEIT(9);
```

```
 writeln(' ORT_','9         ',
         ankunft.STD:2,':',ankunft.MIN:2,'       ',
         '--:--                 --:-- km');
end.
```

Ablaufprotokoll:

```
Bahnhof      Ankunft     Abfahrt   Entf. zur naechsten Station
==============================================================
 ORT_0        --:--       10:00               12.00 km
 ORT_1        10:06       10:08               23.00 km
 ORT_2        10:20       10:22               34.00 km
 ORT_3        10:40       10:42               45.00 km
 ORT_4        11:05       11:07               56.00 km
 ORT_5        11:37       11:39               67.00 km
 ORT_6        12:14       12:16               78.00 km
 ORT_7        12:56       12:58               89.00 km
 ORT_8        13:45       13:47               91.00 km
 ORT_9        14:34       --:--               --.-- --
```

Aufgabe 11: Lagerbestandslisten

Schreiben Sie ein PASCAL–XSC Programm, das mehrere Einzel-Lagerbestandslisten von Filialen einer Kaufhauskette zu einer Gesamtliste komprimiert. Verwenden Sie dazu:

- eine einfach verkettete Liste (*pointer*) mit Elementen bestehend aus den Komponenten `WARENBEZEICHNUNG` (*string* mit max. 20 Zeichen) und `ANZAHL` (0..*maxint*),
- eine Prozedur für die Eingabe einer Lagerbestandsliste,
- einen Operator + für das Zusammenfassen zweier Listen zu einer einzigen durch Addition der beiden `ANZAHL`-Komponenten bei gleichen Warenbezeichnungen bzw. durch Einfügen neuer Listenelemente,
- eine Prozedur für die tabellarische Ausgabe der kompletten Lagerbestandsliste.

Im Hauptprogramm sollen zunächst die Zahl n der Einzellisten und anschließend die n Listen eingelesen werden. Danach sollen die einzelnen Listen mit Hilfe des Operators + zu einer einzigen Liste komprimiert und der Lagerbestand tabellarisch ausgegeben werden.

Lösung:

```
program Listen (input,output);

type
  zeiger = ^ware;
  ware   = record
             warenbezeichnung: string[20];
             anzahl          : 0..maxint;
             next            : zeiger;
           end;

procedure listeneingabe (var liste: zeiger);
var
  h: zeiger;
begin
  liste:= nil;

  repeat
    new (h);
    write ('Warenbezeichnung: ');
    readln;
    read (h^.warenbezeichnung);
    write ('Anzahl: ');
    read (h^.anzahl);
```

```
    h^.next:= liste;
    liste:= h;
  until liste^.anzahl < 0;

  liste:= liste^.next;
end;

operator + (liste1, liste2: zeiger) gesamtliste: zeiger;
var
  gesamt, h1,
  h2, h2vor   : zeiger;
  t           : boolean;
begin
  if (liste1 = nil) then
    gesamt:= liste2
  else
    begin
      gesamt:= liste1;
      h2    := liste2;
      while h2 <> nil do
      begin
        h1:= gesamt;
        repeat
          t:= h1 <> nil;
          if t then
            t:= h1^.warenbezeichnung <> h2^.warenbezeichnung;
          if t then
            h1:= h1^.next;
        until not t;
        if h1 <> nil then
          begin
            h1^.anzahl:= h1^.anzahl + h2^.anzahl;
            liste2    := liste2^.next;
          end
        else
          begin
            liste2   := liste2^.next;
            h2^.next := gesamt;
            gesamt   := h2;
          end;
        h2:= liste2;
      end;
    end;
  gesamtliste:= gesamt;
end;
```

```
procedure ausgabe (liste: zeiger);
var
  h: zeiger;
begin
  h:= liste;
  writeln('Warenbezeichnung ':21,' Anzahl ');
  repeat
    writeln (h^.warenbezeichnung:21, h^.anzahl);
    h:= h^.next;
  until h = nil
end;

var
  n, i          : integer;
  liste, gesamt : zeiger;

begin  {HP}
  gesamt:= nil;
  write('Anzahl der Einzellisten ? ');
  read (n);
  writeln;
  for i:=1 to n do
  begin
    writeln (i:3,'-te Lagerliste  :');
    listeneingabe (liste);
    writeln;
    ausgabe (liste);
    writeln;
    gesamt:= gesamt + liste;
  end;
  writeln;
  writeln(' *** Gesamtliste *** ');
  ausgabe (gesamt);
end.
```

Aufgabe 12: Komplexe Zahlen und Polardarstellung

Eine komplexe Zahl $z = a + ib = (a, b)$ mit $a, b \in I\!R$ kann in Polarkoordinaten dargestellt werden als $z = re^{i\varphi} = (r, \varphi)$ mit $r, \varphi \in I\!R$, $0 \leq \varphi < 2\pi$.

Schreiben Sie ein PASCAL–XSC-Programm, das mit dieser Darstellung arbeitet. Gehen Sie dabei wie folgt vor:

a) Definieren Sie einen geeigneten *record*-Typ *pcomplex* mit den Komponenten *r* und *phi* für die Darstellung von komplexen Zahlen in Polarkoordinaten.

b) Schreiben Sie eine Funktion *pi*, die den Wert von π liefert (Hinweis: $\pi = 4\arctan(1)$).

c) Überladen Sie den Zuweisungsoperator :=, so daß es möglich wird, eine komplexe Zahl $z = a + ib$ vom Typ *complex* an eine Variable vom Typ *pcomplex* mit Komponenten *r* und *phi* zuzuweisen. Die Typ-Wandlung erfolgt nach folgenden Formeln

$$r = \sqrt{a^2 + b^2}$$

$$\varphi = \begin{cases} \pi/2 & \text{für } a = 0 \text{ und } b \geq 0 \\ 3/2 * \pi & \text{für } a = 0 \text{ und } b < 0 \\ \arctan(b/a) & \text{für } a > 0 \text{ und } b \geq 0 \\ 2 * \pi + \arctan(b/a) & \text{für } a > 0 \text{ und } b < 0 \\ \pi + \arctan(b/a) & \text{für } a < 0 \end{cases}$$

Verwenden Sie bei der Berechnung von *r* einen #-Ausdruck zur Erhöhung der Genauigkeit (soweit möglich).

d) Definieren Sie einen Operator $*$ für die Berechnung des Produkts $w = (r, \varphi) = u * v$ von zwei komplexen Zahlen $u = (r_1, \varphi_1)$ und $v = (r_2, \varphi_2)$ vom Typ *pcomplex* in Polardarstellung mit

$$r = r_1 * r_2$$

und

$$\varphi = \begin{cases} \varphi_1 + \varphi_2 & \text{für } \varphi_1 + \varphi_2 < 2\pi \\ \varphi_1 + \varphi_2 - 2 * \pi & \text{sonst} \end{cases}$$

e) Definieren Sie einen Operator / für die Berechnung des Quotienten $w = (r, \varphi) = u/v$ von zwei komplexen Zahlen $u = (r_1, \varphi_1)$ und $v = (r_2, \varphi_2)$ vom Typ *pcomplex* in Polardarstellung mit

$$r = r_1/r_2$$

und

$$\varphi = \begin{cases} \varphi_1 - \varphi_2 & \text{für } \varphi_1 - \varphi_2 \geq 0 \\ \varphi_1 - \varphi_2 + 2 * \pi & \text{sonst} \end{cases}$$

f) Schreiben Sie ein Hauptprogramm, in dem zunächst zwei komplexe Zahlen u und v vom Typ *complex* eingelesen und durch die überladene Zuweisung die entsprechenden Werte *pu* und *pv* erzeugt werden, anschließend die Werte $w = u * v/u/v$ und $pw = pu * pv/pu/pv$ berechnet werden und jeweils der Radius r sowie der Winkel φ von *pw* und von *pw2* $=$ *pcompl*(w) zum Vergleich ausgegeben werden.

Hinweis: Verwenden Sie den Standarddatentyp *complex* und binden Sie das Modul C_ARI ein, das die für diesen Datentyp benötigten Operatoren und Ein-/Ausgabe-Prozeduren zur Verfügung stellt.

Lösung:

```
program polar (input,output);

  use c_ari;

  type
    pcomplex = record
                 r, phi : real;
               end;

  var
    u, v, w         : complex;
    pu, pv, pw, pw2 : pcomplex;

  function pi : real;
    begin
      pi:= 4 * arctan (1);
    end;

  operator := (var pz: pcomplex; z: complex);
    var
      a, b, ph : real;
    begin
      a:= z.re;
      b:= z.im;
      pz.r:= sqrt ( #*(a * a + b * b) );
      if (a = 0) and (b >= 0) then
        ph:= pi/2
      else if (a = 0) and (b < 0) then
        ph:= 3 / 2 * pi
      else if (a > 0) and (b >= 0) then
        ph:= arctan (b/a)
      else if (a > 0) and (b < 0) then
        ph:= 2 * pi + arctan (b/a)
      else
```

```
      ph:= pi + arctan (b/a);
    pz.phi:= ph;
  end;

operator * (u, v : pcomplex) resmul : pcomplex;
  var
    ph : real;
  begin
    resmul.r:= u.r * v.r;
    ph:= u.phi + v.phi;
    if ph < 2 * pi then
      resmul.phi:= ph
    else
      resmul.phi:= ph - 2 * pi;
  end;

operator / (u, v : pcomplex) resdiv : pcomplex;
  var
    ph : real;
  begin
    resdiv.r:= u.r / v.r;
    ph:= u.phi - v.phi;
    if ph >= 0 then
      resdiv.phi:= ph
    else
      resdiv.phi:= ph + 2 * pi;
  end;

begin{Hauptprogramm}
  write ('u eingeben: ');
  read (u);
  write ('v eingeben: ');
  read (v);
  pu := u;
  pv := v;
  w  := u  * v  / u  / v;
  pw2:= w;
  pw := pu * pv / pu / pv;
  writeln ('Radius von pw : ',pw.r);
  writeln ('Winkel von pw : ',pw.phi);
  writeln ('Radius von pw2: ',pw2.r);
  writeln ('Winkel von pw2: ',pw2.phi);
end.
```

Aufgabe 13: Komplexe Division

Der Quotient zweier komplexer Zahlen $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ berechnet sich als

$$\frac{z_1}{z_2} = \frac{z_1\overline{z_2}}{z_2\overline{z_2}} = \frac{(x_1 + iy_1)(x_2 - iy_2)}{{x_2}^2 + {y_2}^2} = \frac{x_1x_2 + y_1y_2}{{x_2}^2 + {y_2}^2} + i\frac{y_1x_2 - x_1y_2}{{x_2}^2 + {y_2}^2}$$

a) Schreiben Sie ein PASCAL–XSC Programm, das die Vereinbarung eines Operators `CDIV` enthält, der diese komplexe Division für zwei komplexe Zahlen vom Standard-Typ *complex* unter Verwendung der Standardoperatoren $+, -, *, /$ für *real*-Zahlen realisiert. Im Hauptprogramm sollen zwei Zahlen vom Typ *complex* eingelesen, dividiert und das Ergebnis ausgegeben werden.

b) Erweitern Sie Ihr Programm dahingehend, daß Sie das Modul `C_ARI` einbinden und nach jedem Einlesen zunächst Ihren Operator `CDIV` und dann den in `C_ARI` vordefinierten Operator / aufrufen und die erhaltenen Werte zum Vergleich ausgeben.

c) Testen Sie dies auch mit den Werten $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ mit

$$\begin{array}{ll} x_1 = 1254027132096, & y_1 = 886731088897 \\ x_2 = 886731088897, & y_2 = 627013566048 \quad . \end{array}$$

Im Imaginärteil werden Sie einen deutlichen Unterschied feststellen.

d) Entwerfen Sie einen weiteren Operator `NCDIV`, der unter Verwendung von *dot-precision*-Ausdrücken bessere Ergebnisse liefert als `CDIV`, und vergleichen Sie anschließend in einem Testlauf die drei Operatoren nochmals.

Lösung:

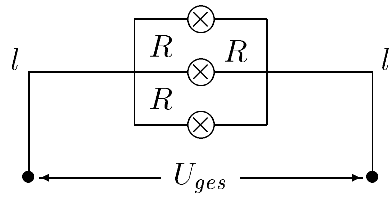
```
program KomplDiv (input,output);

use c_ari;

var z1, z2 : complex;

priority cdiv = *;

operator cdiv (z1, z2 : complex) res : complex;
   var nenner : real;
   begin
      nenner:= sqr(z2.re) + sqr(z2.im);
      res.re:= (z1.re*z2.re + z1.im*z2.im)/nenner;
      res.im:= (z2.re*z1.im - z1.re*z2.im)/nenner;
   end;
```

```
priority ncdiv = *;

operator ncdiv (z1, z2 : complex) nres : complex;
   var nenner : real;
   begin
      nenner := #*(z2.re*z2.re + z2.im*z2.im);
      nres.re:= #*(z1.re*z2.re + z1.im*z2.im)/nenner;
      nres.im:= #*(z2.re*z1.im - z1.re*z2.im)/nenner;
   end;

begin
  write ('z1 = '); read (z1);
  write ('z2 = '); read (z2);
  write ('z1 cdiv  z2 = '); writeln (z1 cdiv z2);
  write ('z1 ncdiv z2 = '); writeln (z1 ncdiv z2);
  write ('z1   /   z2 = '); writeln (z1/z2);
end.
```

Ablaufprotokoll:

```
z_1 = ( 1254027132096, 886731088897 )
z_2 = (  886731088897, 627013566048 )

z1 cdiv  z2 = (  1.414213562373E+00,  0.000000000000E+00 )
z1 ncdiv z2 = (  1.414213562373E+00,  8.478614131949E-25 )
z1   /   z2 = (  1.414213562373E+00,  8.478614131951E-25 )
```

Aufgabe 14: Elektrischer Stromkreis



An einer Spannung $U = 220\text{V}$ sind über eine Leitung der Länge $l = 100\text{m}$ mit Durchmesser $d = 1.5\text{mm}$ und einem spezifischen Widerstand vom Wert $\rho = 0.02857\Omega\text{mm}^2/\text{m}$ ein bis drei Glühbirnen mit einem Widerstand von $R = 240\Omega$ angeschlossen. Dabei sind alle angegebenen Daten mit einem Fehler von 0.5% behaftet.

Für die drei Fälle (1, 2, 3 Glühbirnen angeschlossen) sollen in einem PASCAL–XSC Programm mit Hilfe des Moduls I_ARI und dem Datentyp *interval* die Intervalle berechnet werden, in denen die Werte von R_{ges} (Gesamtwiderstand der Schaltung), I_{ges} (Gesamtstromstärke), U_l (Spannungsanteil an der Leitung) und U_{Gl} (Glühbirnenspannung) schwanken. Es gilt

$$\begin{aligned} R_{ges} &= R_l + R/n \\ I_{ges} &= U/R_{ges} \\ U_l &= R_l \cdot I_{ges} \\ U_{Gl} &= U - U_l \quad . \end{aligned}$$

Dabei soll wie folgt vorgegangen werden:

1) Unter Berücksichtigung der Fehler sollen nach dem Einlesen der o. a. Werte die Intervalle `PI`, `R`, `L`, `D`, `RHO` und `U` berechnet und ausgegeben werden, die die Werte von π, R, l, d, ρ und U einschließen.

2) Danach soll die Einschließung `RL` des Leitungswiderstandes

   $$R_l = (8 \cdot \rho \cdot l)/(\pi \cdot d^2)$$

   berechnet und ausgegeben werden.

3) Schließlich sollen für $n = 1, 2, 3$ Lampen die Intervalle `RGES`, `IGES`, `UL` und `UGL` berechnet und ausgegeben werden.

4) Am Ende soll das Intervall `US` ausgegeben werden, in dem `UGL` für die unterschiedlichen Lampenzahlen insgesamt schwankt.

Hinweis: Eine Intervalleinschließung PI für π berechnet sich durch

$$PI = 4 \cdot \arctan\ ([1,1]).$$

Verwenden Sie den Standarddatentyp *interval* und binden Sie das Modul I_ARI mit den benötigten Intervalloperatoren ein.

Lösung:

```
program Stromkreis (input,output);

 { Elektrischer Stromkreis }

use i_ari;

var
  fehler,
  u, l, d, rho, r, us, pi,
  rl, rges, iges, ul, ugl : interval;
  n                       : integer;

procedure write (var f: text; int: interval;  lang: boolean);
begin
  if lang then
    write (f,'[',int.inf:20:0:-1 ,',',int.sup:20:0:+1 ,']')
    { 13 Mantissenstellen ausgeben }
  else
    write (f,int);
    { Ausgabe wie in I_ARI vorgesehen }
end;

begin
  write ('U   = '); read (u);
  write ('l   = '); read (l);
  write ('d   = '); read (d);
  write ('rho = '); read (rho);
  write ('R   = '); read (r);
  pi := 4 * arctan (intval(1));
  fehler := intval ( (<0.995) , (>1.005) );
  r   := r * fehler;
  l   := l * fehler;
  d   := d * fehler;
  rho := rho * fehler;
  u   := u * fehler;
  writeln;
  writeln ('Intervalle:');
  writeln ('PI   = ',pi : true);
  writeln ('R    = ',r  : true);
  writeln ('L    = ',l  : true);
  writeln ('D    = ',d  : true);
  writeln ('RHO  = ',rho: true);
  writeln ('U    = ',u  : true);
  writeln;
  rl:= (8*rho*l)/(pi*sqr(d));
  writeln ('Einschliessung des Leitungswiderstandes:');
```

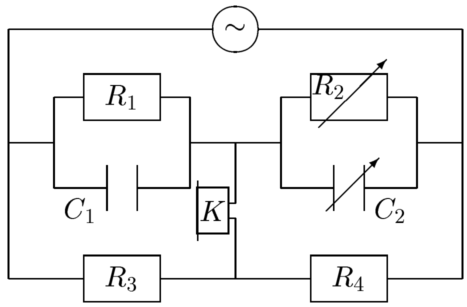
```
  writeln ('Rl   = ',rl : true);
  for n:= 1 to 3 do
  begin
    writeln;
    rges:= rl + r/n;
    iges:= u / rges;
    ul  := rl * iges;
    ugl := u - ul;
    if n = 1 then
      us:= ugl
    else
      us:= us +* ugl;
    write('Mit ',n:1,' Gluehbirne');
    if n <> 1 then write ('n');
    writeln(' ergeben sich die folgenden Einschliessungen fuer');
    writeln('- Gesamtwiderstand:    Rges = ',rges : true);
    writeln('- Gesamtstromstaerke:  Iges = ',iges : true);
    writeln('- Leitungsspannung:    Ul   = ',ul   : true);
    writeln('- Gluehbirnenspannung: Ugl  = ',ugl  : true);
    write('weiter mit return'); readln; writeln;
  end;
  writeln ('Die Gluehbirnenspannung Ugl schwankt also insgesamt im ');
  writeln ('Intervall  ',us : true);
end.
```

Ablaufprotokoll:

```
U   = 220
l   = 100
d   = 1.5
rho = 0.02857
R   = 240

Intervalle:

PI   = [ 3.141592653589E+00, 3.141592653590E+00]
R    = [ 2.388000000000E+02, 2.412000000000E+02]
L    = [ 9.950000000000E+01, 1.005000000000E+02]
D    = [ 1.492500000000E+00, 1.507500000000E+00]
RHO  = [ 2.842715000000E-02, 2.871285000000E-02]
U    = [ 2.189000000000E+02, 2.211000000000E+02]

Einschliessung des Leitungswiderstandes:
Rl   = [ 3.169435182648E+00, 3.298783385818E+00]

Mit 1 Gluehbirne ergeben sich die folgenden Einschliessungen fuer
- Gesamtwiderstand:    Rges = [ 2.419694351826E+02, 2.444987833859E+02]
```

```
- Gesamtstromstaerke:  Iges = [ 8.953009784695E-01, 9.137517713060E-01]
- Leitungsspannung:    Ul   = [ 2.837598420220E+00, 3.014269161947E+00]
- Gluehbirnenspannung: Ugl  = [ 2.158857308380E+02, 2.182624015798E+02]

Mit 2 Gluehbirnen ergeben sich die folgenden Einschliessungen fuer
- Gesamtwiderstand:    Rges = [ 1.225694351826E+02, 1.238987833859E+02]
- Gesamtstromstaerke:  Iges = [ 1.766764725350E+00, 1.803875490416E+00]
- Leitungsspannung:    Ul   = [ 5.599646279985E+00, 5.950594497869E+00]
- Gluehbirnenspannung: Ugl  = [ 2.129494055021E+02, 2.155003537201E+02]

Mit 3 Gluehbirnen ergeben sich die folgenden Einschliessungen fuer
- Gesamtwiderstand:    Rges = [ 8.276943518264E+01, 8.369878338582E+01]
- Gesamtstromstaerke:  Iges = [ 2.615330726982E+00, 2.671275930688E+00]
- Leitungsspannung:    Ul   = [ 8.289121220357E+00, 8.811960659090E+00]
- Gluehbirnenspannung: Ugl  = [ 2.100880393409E+02, 2.128108787797E+02]

Die Gluehbirnenspannung Ugl schwankt also insgesamt im
Intervall  [ 2.100880393409E+02, 2.182624015798E+02]
```

Aufgabe 15: Wechselstrom-Meßbrücke



Die unbekannte Kapazität C_1 und der unbekannte Widerstand R_1 können mit Hilfe der oben angegebenen Schaltung bestimmt werden. Man variiert dazu den Kondensator C_2 und den Widerstand R_2 solange, bis der Ton im Lautsprecher K ein Minimum erreicht oder verschwindet. In diesem Fall gilt

$$\begin{aligned} C_1 &= R_4 \cdot C_2 / R_3 \\ R_1 &= R_3 \cdot R_2 / R_4 \quad . \end{aligned}$$

Für die Werte von R_3 und R_4 gilt nach den Herstellerangaben:

$$\begin{array}{rcccr} 9.9\Omega & \leq & R_3 & \leq & 10.1\Omega \\ 6.8\Omega & \leq & R_4 & \leq & 6.9\Omega \quad , \end{array}$$

für C_2 und R_2 gelten, bedingt durch Meßungenauigkeiten, folgende Abschätzungen:

$$\begin{array}{rcccr} 40.2\text{mF} & \leq & C_2 & \leq & 41.5\text{mF} \\ 18.3\Omega & \leq & R_2 & \leq & 19.8\Omega \quad . \end{array}$$

Schreiben Sie ein PASCAL–XSC Programm, das

- die Grenzwerte von R_3, R_4, C_2 und R_2 einliest,
- die Intervalleinschließungen von C_1 und R_1 berechnet und ausgibt und
- außerdem Einschließungen für C_1 und R_1 unter der Annahme, daß der Fehler bei C_2 und R_2 um 10% höher liegt, berechnet und ausgibt.

Hinweis: Die Berücksichtigung eines um 10% größeren Fehlers bei den Intervallen `C2` und `R2` sollte dadurch erfolgen, daß die Intervalle um 10% des halben Durchmessers nach oben und unten vergrößert werden.

Lösung:

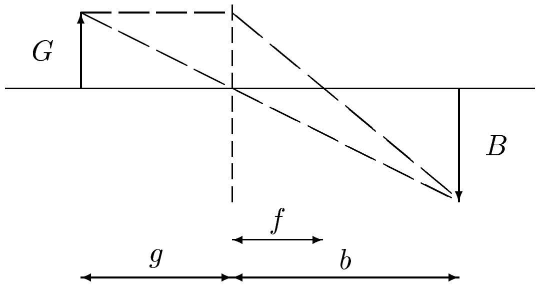
```
program Messbruecke (input,output);
use i_ari;
var
  c1, c2, r1, r2, r3, r4 : interval;
  d : real;

procedure write (var f: text; int: interval;  lang: boolean);
begin
  if lang then
    write (f,'[',int.inf:20:0:-1 ,',',int.sup:20:0:+1 ,']')
    { 13 Mantissenstellen ausgeben }
  else
    write (f,int);
    { Ausgabe wie in I_ARI vorgesehen }
end;

begin
  write('untere Schranke fuer R3: '); read(r3.inf:-1);
  write('obere  Schranke fuer R3: '); read(r3.sup:+1);
  write('untere Schranke fuer R4: '); read(r4.inf:-1);
  write('obere  Schranke fuer R4: '); read(r4.sup:+1);
  write('untere Schranke fuer C2: '); read(c2.inf:-1);
  write('obere  Schranke fuer C2: '); read(c2.sup:+1);
  write('untere Schranke fuer R2: '); read(r2.inf:-1);
  write('obere  Schranke fuer R2: '); read(r2.sup:+1);
  c1:= r4 * c2 / r3;
  r1:= r3 * r2 / r4;
  writeln;
  writeln('C1 = ', c1 : true);
  writeln('R1 = ', r1 : true);
  d := diam(c2) /> 20;
  c2:= intval ( c2.inf -< d , c2.sup +> d );
  d := diam(r2) /> 20;
  r2:= intval ( r2.inf -< d , r2.sup +> d );
  c1:= r4 * c2 / r3;
  r1:= r3 * r2 / r4;
  writeln;
  writeln('Ergebnisse mit 10% hoeherem Fehler bei C2 und R2:');
  writeln('C1 = ', c1 : true);
  writeln('R1 = ', r1 : true);
end.
```

Ablaufprotokoll:

```
C1 = [ 2.706534653465E+01, 2.892424242425E+01]
R1 = [ 2.625652173913E+01, 2.940882352942E+01]

Ergebnisse mit 10% hoeherem Fehler bei C2 und R2:
C1 = [ 2.702158415841E+01, 2.896954545455E+01]
R1 = [ 2.614891304347E+01, 2.952022058824E+01]
```

Bemerkung: Diese Aufgabe demonstriert die, bedingt durch das Operatorkonzept, einfache Anwendung der Intervallarithmetik bei Fehlerrechnungen in der Technik.

Aufgabe 16: Optische Linse



Mit einer Linse mit Brennweite $f = (20 \pm 1)$cm wurde mit dem Bild B eines Gegenstands G eine Bildweite $b = (25 \pm 1)$cm gemessen. Die Abbildungsgleichung für dünne Linsen zur Ermittlung der Gegenstandsweite g lautet

$$\frac{1}{f} = \frac{1}{b} + \frac{1}{g} \quad .$$

Für g ergibt sich dann die Gleichung

$$g = \frac{1}{\frac{1}{f} - \frac{1}{b}} \quad .$$

Üblicherweise wird der Wert $g = g_0 \pm \triangle g$ mit einem Fehlerterm $\triangle g$ dadurch ermittelt, daß man zunächst

$$g_0 = \frac{1}{\frac{1}{f_0} - \frac{1}{b_0}}$$

und anschließend

$$\triangle g = \frac{\triangle f}{(1 - \frac{f_0}{b_0})^2} + \frac{\triangle b}{(\frac{b_0}{f_0} - 1)^2}$$

berechnet. Dabei ist $f_0 = 20$cm, $b_0 = 25$cm und $\triangle f = \triangle b = 1$cm. Schreiben Sie ein PASCAL–XSC Programm, das

- die Werte für f_0, b_0, $\triangle f$ und $\triangle b$ einliest,
- das Intervall $g = g_0 \pm \triangle g$ nach der oben beschriebenen Methode berechnet,
- das Intervall g aus den Intervallen f und b mit Intervallrechnung gemäß

  $$g = \frac{1}{\frac{1}{f} - \frac{1}{b}}$$

  berechnet und

- f, b und die zwei Werte von g mit entsprechenden Kommentaren ausgibt.

Stellen Sie anhand der intervallmäßig berechneten Einschließung für g fest, ob die üblicherweise benutzte erste Methode ein richtiges Ergebnis liefert.

Lösung:

```
program opt_lens (input,output);

use i_ari;

var
  g0, dg, f0, df, b0, db : real;
  g, f, b                : interval;

procedure write (var f: text; int: interval;  lang: boolean);
begin
  if lang then
    write (f,'[',int.inf:20:0:-1 ,',',int.sup:20:0:+1 ,']')
    { 13 Dezimalstellen ausgeben }
  else
    write (f,int);
    { Ausgabe wie in I_ARI vorgesehen }
end;

begin
  write('f0 = '); read(f0);
  write('df = '); read(df);
  write('b0 = '); read(b0);
  write('db = '); read(db);
  writeln;
  f:= intval (f0 - df , f0 + df);
  b:= intval (b0 - db , b0 + db);
  writeln ('f = ', f : true);
  writeln ('b = ', b : true);
  writeln;
  g0 := 1 / (1/f0 - 1/b0);
  dg := df / sqr(1 - f0/b0) + db / sqr(b0/f0 - 1);
  g  := intval (g0 - dg , g0 + dg);
  writeln ('g = g0 +/- dg       = ', g : true);
  g:= 1/(1/f - 1/b);
  writeln ('g = 1 / (1/f - 1/b) = ', g : true);
end.
```

Ablaufprotokoll:

```
f0 = 20
df = 1
b0 = 25
db = 1

f = [ 1.900000000000E+01, 2.100000000000E+01]
b = [ 2.400000000000E+01, 2.600000000000E+01]

g = g0 +/- dg        = [ 5.900000000000E+01, 1.410000000000E+02]
g = 1 / (1/f - 1/b) = [ 7.057142857137E+01, 1.680000000004E+02]
```

Bemerkung: Die im Ablaufprotokoll aufgezeigten Ergebnisse zeigen, daß die üblicherweise verwendete Methode zur Fehlerabschätzung ein falsches Intervall berechnet.

Hinweis: Die Aufgaben 14, 15 und 16 wurden angeregt durch den Beitrag *Technical Calculations by means of Interval Mathematics* von P. Thieler im Tagungsband von K. Nickel (Ed.): *Interval Mathematics 1985*, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1986.

Aufgabe 17: Intervallauswertung eines Polynoms

Schreiben Sie ein PASCAL–XSC Programm unter Verwendung des Moduls I_ARI zur intervallmäßigen Berechnung des Polynomwertes

$$p(X) = 1 + 3X - 10X^2 \quad .$$

Vergleichen Sie die Ergebnisse für folgende Darstellungen (mit X vom Typ *interval*):

1) $(1 - 2 * X) * (1 + 5 * X)$
2) $1 + 3 * X - 10 * \mathrm{sqr}(X)$
3) $1 + X * (3 - 10 * X)$ (Hornerschema)
4) $1 + 3 * m(X) - 10 * \mathrm{sqr}(m(X)) + (3 - 20 * X)(X - m(X))$
   (Mittelwertform, mit $m(X)$ Mittelpunkt von X)

Wählen Sie als Beispiele für X sowohl enge Intervalle (Weite etwa eine Einheit in der 12. Dezimalstelle) als auch Intervalle mit anderen Durchmessern. Dabei sollen auch Intervalle um die Nullstellen ($x = 0.5$ und $x = -0.2$) und um das Extremum ($x = 0.15$) berücksichtigt werden.

Hinweis: Verwenden Sie für die Berechnung des Mittelpunkts eines Intervalls einen #-Ausdruck zur Erzielung maximaler Genauigkeit.

Lösung:

```
program int_poly (input, output);

use i_ari;

var x : interval;

function m (x : interval) : real;

  begin
    m:= #* (0.5 * x.inf + 0.5 * x.sup);
  end;

begin
  repeat
    write ('X eingeben: '); read(x);
    writeln;
    writeln ('Methode 1: p(X) = ', (1-2*x)*(1+5*x));
    writeln ('Methode 2: p(X) = ', 1+3*x-10*sqr(x));
    writeln ('Methode 3: p(X) = ', 1+x*(3-10*x));
    writeln ('Methode 4: p(X) = ',
                         1+3*m(x)-10*sqr(m(x))+(3-20*x)*(x-m(x)));
    writeln; writeln;
  until x = 0;
end.
```

Ablaufprotokoll:

```
X eingeben: [0.5,0.5]

Methode 1: p(X) = [  0.000000000000E+00,  0.000000000000E+00]
Methode 2: p(X) = [  0.000000000000E+00,  0.000000000000E+00]
Methode 3: p(X) = [  0.000000000000E+00,  0.000000000000E+00]
Methode 4: p(X) = [  0.000000000000E+00,  0.000000000000E+00]

X eingeben: [0.4999999999,0.5]

Methode 1: p(X) = [             0.0E+00,             7.0E-10]
Methode 2: p(X) = [            -3.0E-10,             1.0E-09]
Methode 3: p(X) = [             0.0E+00,             7.0E-10]
Methode 4: p(X) = [             0.0E+00,             7.0E-10]

X eingeben: [-0.2000000000001,-0.1999999999999]

Methode 1: p(X) = [            -1.5E-12,             7.1E-13]
Methode 2: p(X) = [            -8.0E-13,             7.0E-13]
Methode 3: p(X) = [            -1.0E-12,             7.0E-13]
Methode 4: p(X) = [            -7.1E-13,             7.1E-13]

X eingeben: [0.1499999999999,0.1500000000001]

Methode 1: p(X) = [  1.224999999998E+00,  1.225000000002E+00]
Methode 2: p(X) = [  1.224999999998E+00,  1.225000000002E+00]
Methode 3: p(X) = [  1.224999999999E+00,  1.225000000001E+00]
Methode 4: p(X) = [  1.224999999999E+00,  1.225000000001E+00]


X eingeben: [0.1,0.2]

Methode 1: p(X) = [             9.0E-01,             1.6E+00]
Methode 2: p(X) = [             9.0E-01,             1.5E+00]
Methode 3: p(X) = [             1.1E+00,             1.4E+00]
Methode 4: p(X) = [             1.1E+00,             1.3E+00]
```

Aufgabe 18: Intervall-Matrixrechnung

Gegeben sind die Intervall-Matrizen A und B mit

$$A = \begin{pmatrix} [1,1] & [0,1] \\ [1,1] & [-1,1] \end{pmatrix}, \quad B = \begin{pmatrix} [-1,2] & [3,4] \\ [2,2] & [-6,-4] \end{pmatrix}.$$

In einem PASCAL–XSC Programm soll

a) berechnet werden: $A + B$, $A - B$, $A \cdot B$,

b) durch Rechnung gezeigt werden: $A \cdot (A \cdot A) \neq (A \cdot A) \cdot A$,

c) durch Rechnung gezeigt werden: $A \cdot (B + A) \overset{\subset}{\neq} A \cdot B + A \cdot A$.

Hinweis: Verwenden Sie die im Modul MVI_ARI zur Verfügung gestellten Operatoren.

Lösung:

```
program int_matr (input,output);

use i_ari, mvi_ari;

var
  a, b, c : imatrix[1..2,1..2];

begin
  a[1,1]:= 1;
  a[1,2]:= intval(0,1);
  a[2,1]:= 1;
  a[2,2]:= intval(-1,1);

  b[1,1]:= intval(-1,2);
  b[1,2]:= intval(3,4);
  b[2,1]:= 2;
  b[2,2]:= intval(-6,-4);

  writeln ('A = '); writeln;
  writeln (a);
  writeln ('B = '); writeln;
  writeln (b);
  writeln ('A + B = '); writeln;
  writeln (a+b);
  writeln ('A - B = '); writeln;
  writeln (a-b);
  writeln ('A * B = '); writeln;
  writeln (a*b);
```

```
  writeln ('A * (A * A) = '); writeln;
  writeln (a*(a*a));
  writeln ('(A * A) * A = '); writeln;
  writeln ((a*a)*a);
  writeln ('A * (B + A) = '); writeln;
  writeln (a*(b+a));
  writeln ('A * B + A * A = '); writeln;
  writeln (a*b+a*a);
end.
```

Aufgabe 19: Automatische Differentiation

Mit Hilfe der sogenannten Differentiationsarithmetik (vgl. [31]) sollen die Werte der Funktion

$$f(x) = x \cdot \frac{4+x}{3-x}$$

und die Werte ihrer Ableitung $f'(x)$ im Bereich $-4 \leq x \leq 2$ an den Stützstellen $x_i = -4 + ih$, $i = 0, \ldots, 50$ mit $h = 0.12$ berechnet werden.

Die Differentiationsarithmetik ist eine Arithmetik geordneter Paare der Form

$$U = (u, u') \quad \text{mit} \quad u, u' \in I\!R.$$

In der ersten Komponente von U steht der Funktionswert, in der zweiten der Wert der Ableitung. Die Regeln für die Arithmetik lauten dann

$$\begin{array}{lclcl} U+V &=& (u,u')+(v,v') &=& (u+v, u'+v') \\ U-V &=& (u,u')-(v,v') &=& (u-v, u'-v') \\ U*V &=& (u,u')*(v,v') &=& (u*v, u*v'+u'*v) \\ U/V &=& (u,u')/(v,v') &=& (u/v, (u'-u*v'/v)/v), \quad v \neq 0, \end{array}$$

wobei in der zweiten Komponente jeweils die entsprechende Differentiationsregel verwendet wird. Für die unabhängige Variable x und eine beliebige Konstante c folgt wegen $\frac{dx}{dx} = 1$ und $\frac{dc}{dx} = 0$

$$X = (x, 1) \quad \text{und} \quad C = (c, 0).$$

Vereinbart man in einem PASCAL–XSC Programm einen Typ `DIFF` als *record* von zwei *real*-Werten und verwendet man in der Funktion f mit Argument und Ergebnis vom Typ `DIFF` die oben beschriebenen Operatoren $+, -, *, /$ der Differentiationsarithmetik, also

$$f(X) = X * ((4,0) + X)/((3,0) - X),$$

so erhält man wegen

$$f(X) = f((x,1)) = (f(x), f'(x))$$

zusätzlich zum Funktionswert automatisch den Wert der Ableitung mitberechnet. Schreiben Sie ein PASCAL–XSC Modul mit

a) einer Typvereinbarung `DIFF`

b) Vereinbarungen der Operatoren $+, -, *, /$ gemäß den obigen Regeln für die Differentiationsarithmetik.

Schreiben Sie dann ein PASCAL–XSC Programm mit

a) einer Funktion F, die die Operatoren des Moduls benutzt und damit sowohl den Wert der oben genannten Funktion f als auch den automatisch mitberechneten Wert ihrer Ableitung liefert und

b) einem Hauptprogramm, das in einer Schleife mittels der Funktion F die Werte von $f(x)$ und $f'(x)$ an den Stützstellen berechnet und tabelliert.

Hinweis: Die Konstanten 4 bzw. 3 in der Funktion f haben als `DIFF`-Variablen die Darstellung $(4,0)$ bzw. $(3,0)$. Die x_i werden als $(x_i, 1)$ dargestellt.

Lösung:

```
module diffari;

global type diff = record
                     f, df : real;
                   end;

global operator := (var a: diff; r: real);
  begin
    a.f := r;
    a.df:= 0;
  end;

global operator + (a,b: diff) respl: diff;
  begin
    respl.f  := a.f + b.f;
    respl.df := a.df + b.df;
  end;

global operator - (a,b: diff) resmi: diff;
  begin
    resmi.f  := a.f - b.f;
    resmi.df := a.df - b.df;
  end;

global operator * (a,b: diff) resmu: diff;
  begin
    resmu.f  := a.f * b.f;
    resmu.df := a.f * b.df + a.df * b.f;
  end;

global operator / (a,b: diff) resdi: diff;
  begin
    resdi.f  := a.f / b.f;
    resdi.df := (a.df - a.f * b.df / b.f) / b.f ;
  end;

end.
```

```
program autodiff (input,output);

use diffari;

function f (x: diff): diff;
  var
    vier, drei : diff;
  begin
    vier := 4;
    drei := 3;
    f:= x*((vier+x)/(drei-x));
  end;

var
  x,y : diff;
  h   : real;
  i   : integer;

begin
  x.df := 1;
  h    := 0.12;
  writeln('x          ' : 19,'f(x)        ' : 19,'f''(x)      ');
  for i:= 0 to 50 do
  begin
    x.f := -4 + i * h;
    y   := f(x);
    writeln (x.f,'    ',y.f,'    ',y.df);
  end;
end.
```

Ablaufprotokoll:

```
         x                    f(x)                    f'(x)
============================================================
-4.000000000000E+00    0.000000000000E+00   -5.714285714284E-01
-3.880000000000E+00   -6.767441860467E-02   -5.563480259601E-01
-3.760000000000E+00   -1.334911242603E-01   -5.404572669023E-01
-3.640000000000E+00   -1.973493975904E-01   -5.236971984324E-01
-3.520000000000E+00   -2.591411042945E-01   -5.060032368550E-01
-3.400000000000E+00   -3.187500000000E-01   -4.873046875000E-01
-3.280000000000E+00   -3.760509554141E-01   -4.675240374864E-01
-3.160000000000E+00   -4.309090909090E-01   -4.465761511220E-01
-3.040000000000E+00   -4.831788079470E-01   -4.243673523091E-01
-2.920000000000E+00   -5.327027027026E-01   -4.007943754564E-01
-2.800000000000E+00   -5.793103448275E-01   -3.757431629013E-01
-2.680000000000E+00   -6.228169014085E-01   -3.490874826423E-01
-2.560000000000E+00   -6.630215827338E-01   -3.206873350239E-01
-2.440000000000E+00   -6.997058823531E-01   -2.903871107264E-01
-2.320000000000E+00   -7.326315789473E-01   -2.580134546892E-01
-2.200000000000E+00   -7.615384615384E-01   -2.233727810652E-01
-2.080000000000E+00   -7.861417322834E-01   -1.862483724970E-01
-1.960000000000E+00   -8.061290322580E-01   -1.463969823102E-01
-1.840000000000E+00   -8.211570247933E-01   -1.035448398335E-01
-1.720000000000E+00   -8.308474576271E-01   -5.738293593820E-02
...
```

Aufgabe 20: Newton-Verfahren mit automat. Differentiation

Die Nullstelle einer Funktion $f(x)$ läßt sich mit Hilfe des Newton-Verfahrens und einem geeignet gewählten x_0 folgendermaßen bestimmen:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots$$

Mittels des in der letzten Aufgabe entworfenen Moduls können bei entsprechender Verwendung der Operatoren in der Funktion f der Funktionswert und der Wert der Ableitung simultan berechnet werden.

Schreiben Sie ein PASCAL–XSC Programm, das mit Hilfe der Differentiationsarithmetik das oben beschriebene Newton-Verfahren realisiert. Verwenden Sie die Funktion

$$f(x) = e^x - x - 5$$

und führen Sie, beginnend mit einem einzulesenden Startwert x_0, fünf Iterationsschritte durch. Die x_i, $i = 1, \ldots, 5$ sind jeweils auszugeben.

Testen Sie Ihr Programm mit den Startwerten $x_0 = 2.0$ und $x_0 = -5.0$. Die Nullstellen liegen bei $x = 1.9368470722$ und $x = -4.99321618865$.

Hinweis: Für die Variable $U = (u, u')$ vom Typ `DIFF` läßt sich die Funktion e^U implementieren als

$$e^U = e^{(u,u')} = (e^u, u' * e^u).$$

Lösung:

```
program newt_diff (input,output);

use diffari;

function exp (x: diff) : diff;
  begin
    exp.f := exp(x.f);
    exp.df:= x.df * exp(x.f);
  end;

function f (x: diff) : diff;
  var
    fuenf: diff;
  begin
    fuenf := 5;
    f:= exp(x) - x - fuenf;
  end;
```

```
var
  x, y : diff;
  i    : integer;

begin
  write ('x0 : '); read (x.f);
  x.df := 1;
  for i:= 1 to 5 do
  begin
    y   := f(x);
    x.f := x.f - y.f/y.df;
    writeln ('x',i:1,' : ',x.f);
  end;
end.
```

Ablaufprotokoll:

```
x0 :  2.0
x1 :  1.939105856498E+00
x2 :  1.936850383715E+00
x3 :  1.936847407225E+00
x4 :  1.936847407220E+00
x5 :  1.936847407220E+00

x0 : -5.0
x1 : -4.993216345094E+00
x2 : -4.993216188648E+00
x3 : -4.993216188648E+00
x4 : -4.993216188648E+00
x5 : -4.993216188648E+00
```

Aufgabe 21: Zeitrechnung

Eine in Stunden (h), Minuten (m) und Sekunden (s) zerlegte Zeitangabe zwischen 00.00.00 Uhr und 23.59.59 Uhr werde durch einen *record*-Typ `UHR` dargestellt. Schreiben Sie in PASCAL–XSC eine Vereinbarung für einen Operator +, der zwei solche Zeitangaben addiert und gegebenenfalls 24 h subtrahiert, damit das Ergebnis wieder im Typ `UHR` darstellbar ist. Verwenden Sie diese Vereinbarung in einem Programm, das bis zu *maxint* Zeitangaben einliest, aufaddiert und die Gesamtzeit jedes Mal ausgibt. Die Eingabeschleife werde bei Eingabe von 0.00.00 abgebrochen.

Die Eingabe der Uhrzeiten soll als *string* in der Form `hh.mm.ss` erfolgen. Dieser Eingabe-*string* soll dann mit den *string*-Funktionen von PASCAL–XSC in den Typ `UHR` gewandelt werden. Bei der Ausgabe ist umgekehrt zu verfahren, d. h. nach Wandlung von `UHR` nach *string* soll die Zeit wieder in der Form `hh.mm.ss` ausgegeben werden.

Lösung:

```
program zeitrech (input,output);
type UHR = record
             h : 0..23;
             m : 0..59;
             s : 0..59;
           end;

var a, b        : string[8];
    zeit, time  : UHR;
    i           : integer;

operator + (a, b: UHR) summe : UHR;
var hilf : 0..119;
begin
  hilf    := a.s + b.s;
  summe.s := hilf mod 60;
  hilf    := a.m + b.m + hilf div 60;
  summe.m := hilf mod 60;
  hilf    := a.h + b.h + hilf div 60;
  summe.h := hilf mod 24
end;

begin
  time.h := 0;
  time.m := 0;
  time.s := 0;
  i:= 0;
  repeat
    i:= succ (i);
    write ('Bitte Zeitdifferenz angeben : ');
```

```
    readln;
    read (a);
    zeit.h := ival(substring(a,1,2));
    zeit.m := ival(substring(a,4,2));
    zeit.s := ival(substring(a,7,2));
    time   := time + zeit;
    b      := image (time.h,2) + '.' + image (time.m,2)
              + '.' + image (time.s,2);
    writeln ('Neue Zeit                    : ', b);
  until (i = maxint) or (zeit.h+zeit.m+zeit.s = 0);
end.
```

Aufgabe 22: Iterationsverfahren

Das Vektoriterationsverfahren

$$(*) \quad x^{(k+1)} = c + Ax^{(k)}, \quad k = 0, 1, 2, \ldots$$

mit $c, x^{(k)} \in I\!R^n$, $k = 0, 1, 2, \ldots$ und $A \in I\!R^{n \times n}$ sei für die auf dem Rechner zu bearbeitenden Probleme konvergent, d. h. der Spektralradius von A sei kleiner als 1.

Schreiben Sie ein PASCAL–XSC-Programm für die Durchführung dieses Verfahrens. Entwerfen Sie dazu zunächst ein Modul *matvek*, das die benötigten Typen, Operatoren und Prozeduren zur Verfügung stellt. Dieses Modul soll folgende Teile enthalten:

a) eine dynamische Typvereinbarung *vektor*, worin ein Vektor als eindimensionales Feld mit Komponententyp *real* festgelegt wird,

b) eine dynamische Typvereinbarung *matrix*, worin eine Matrix als zweidimensionales Feld mit Komponententyp *real* festgelegt wird,

c) einen Operator = für den Vergleich zweier Vektoren $a = (a_i)$ und $b = (b_i)$ gemäß

$$a = b \quad \Leftrightarrow \quad a_i = b_i \quad \text{für alle } i$$

d) einen Operator + für die Addition zweier Vektoren $a = (a_i)$ und $b = (b_i)$ gemäß

$$c = a + b \quad \text{mit} \quad c_i = a_i + b_i, \quad \text{für alle } i$$

e) einen Operator $*$ für die Multiplikation einer Matrix $A = (a_{ij})$ mit einem Vektor $x = (x_i)$ gemäß

$$y = A * x \quad \text{mit} \quad y_i = \sum_j a_{ij} x_j, \quad \text{für alle } i$$

unter Verwendung des Datentyps *dotprecision*, so daß bei der Berechnung von y_i nur einmal gerundet wird,

f) eine Überladung der Prozedur *read* für die Eingabe eines Vektors,

g) eine Überladung der Prozedur *read* für die Eingabe einer Matrix,

h) eine Überladung der Prozedur *write* für die Ausgabe eines Vektors.

Entwerfen Sie dann ein Programm *iteration*, welches das Modul *matvek* verwendet und die folgenden Teile enthält:

1) eine Prozedur *main* mit formalem Parameter n, in der mit Hilfe der Typen, Prozeduren und Operatoren aus dem Modul *matvek* die für die Iteration benötigten Variablen als n-dimensionale Vektoren bzw. Matrizen vereinbart werden (Indexbereich $1, \ldots, n$), c, A und $x^{(0)}$ eingelesen werden, die Iteration $(*)$ solange ausgeführt wird, bis entweder $x^{(k+1)} = x^{(k)}$ oder $k = 20$ erfüllt ist, und abschließend der zuletzt berechnete Vektor $x^{(k+1)}$ ausgegeben wird,

2) ein Hauptprogramm, in dem die Dimension n eingelesen und die Prozedur *main* aufgerufen wird.

Lösung:

```
module matvek;

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

  global operator = (a, b: vektor) equ: boolean;
    var
      i : integer;
    begin
      i:= ubound(a) - 1;
      repeat
        i:= i + 1;
      until (a[i] <> b[i]) or (i = ubound(a));
      equ:= (a[i] = b[i]);
    end;

  global operator + (a,b: vektor) vadd: vektor[lbound(a)..ubound(a)];
    var
      i : integer;
    begin
      for i:= lbound(a) to ubound(a) do
        vadd[i] := a[i] + b[i];
    end;

  global operator * (A: matrix; x: vektor)
                   mvmul: vektor[lbound(x)..ubound(x)];
    var
      i, j : integer;
      d    : dotprecision;
    begin
      for i:= lbound(A) to ubound(A) do
      begin
        d:= #(0);
        for j:= lbound(A,2) to ubound(A,2) do
```

```
          d:= #(d + a[i,j] * x[j]);
        mvmul[i]:= #*(d);
      end;
    end;

  global procedure read (var f: text; var c: vektor);
    var
      i : integer;
    begin
      for i:= lbound(c) to ubound(c) do
        read (f, c[i]);
    end;

  global procedure read (var f: text; var A: matrix);
    var
      i, j : integer;
    begin
      for i:= lbound(A) to ubound(A) do
        for j := lbound(A,2) to ubound(A,2) do
          read(f, A[i,j]);
    end;

  global procedure write (var f: text; c: vektor);
    var
      i : integer;
    begin
      for i:= lbound(c) to ubound(c) do
        writeln (f, c[i]);
    end;

end.
```

```
program iterate (input,output);

  use matvek;

  var
    n : integer;

  procedure main (n: integer);
    var
      i, j, k       : integer;
      c, xk, xk1, y : vektor[1..n];
      A             : matrix[1..n,1..n];
    begin
      writeln ('Vektor c eingeben');
      read (c);
      writeln ('Matrix A eingeben');
      read (a);
      writeln ('Vektor x0 eingeben');
      read (xk1);

      {Iteration}
      k:= -1;
      repeat
        xk:= xk1;
         k:= k + 1;
        xk1 := c + A * xk;
      until (xk1 = xk) or (k = 20);
      writeln ('Letzte Iterierte: ');
      write (xk1);
    end;

begin {Hauptprogramm}
  write ('n = ');
  read (n);
  main (n);
end.
```

Aufgabe 23: Spur einer Produktmatrix

Die Spur einer $n \times n$-Matrix $A = (a_{ij})$ ist definiert durch

$$SPUR := \sum_{i=1}^{n} a_{ii} = a_{11} + \cdots + a_{nn}$$

also als die Summe der Diagonalelemente. Schreiben Sie ein PASCAL–XSC Programm, das die Dimension n und anschließend zwei $n \times n$-Matrizen A und B einliest, die Spur der Produktmatrix $C = A \cdot B$ berechnet und den Wert ausgibt.

Verwenden Sie das Modul MV_ARI, in dem die Prozeduren und Operatoren für die dynamischen Typen *rvector* und *rmatrix* vereinbart sind, und entwerfen Sie eine Funktion `SPUR1`, die die Spur des Produkts zweier Matrizen mit herkömmlicher Rechnung ermittelt und eine Funktion `SPUR2`, die für die maximal genaue Berechnung dieser Spur einen #-Ausdruck verwendet. Vergleichen Sie die Ergebnisse anhand von Beispielen.
Testen Sie Ihr Programm auch mit den Matrizen

$$A = \begin{pmatrix} 10^9 & 8 & 126 & -237 \\ 100 & 2 & -12 & 1 \\ 10^5 & 10 & -10^7 & 81 \\ 13 & -3 & 30 & 10^{-7} \end{pmatrix}, \quad B = \begin{pmatrix} 10^8 & 85 & 8 & 6 \\ 12 & 3 & 10^3 & 156 \\ 3 & 14 & 10^{10} & 13 \\ 2 & -8332 & -10^4 & -10^{-8} \end{pmatrix}.$$

Lösung:

```
program spur (input, output);

use  mv_ari;

var  n: integer;

function spur1 (a, b: rmatrix): real;
  var  i: integer;
       s: real;
  begin
    s:= 0;
    for i:=lb(a,1) to ub(a,1) do
      s := s + a[i] * rvector(b[*,i]);
    spur1:= s;
  end;

function spur2 (a, b: rmatrix): real;
  var  i: integer;
  begin
    spur2 := #* (for i:=lb(a,1) to ub(a,1) sum (a[i]*rvector(b[*,i])))
  end;
```

```
procedure main (n: integer);

  var
    a, b     : rmatrix[1..n,1..n];
    sp1, sp2 : real;

  begin
    writeln;
    writeln('Matrix A eingeben');
    read (a);
    writeln;
    writeln('Matrix B eingeben');
    read (b);
    writeln;
    sp1:= spur1 (a,b);
    sp2:= spur2 (a,b);
    writeln('Spur von A*B herkoemmlich berechnet  : ',sp1);
    writeln('Spur von A*B maximal genau berechnet : ',sp2);
  end;

begin
  write('Dimension n eingeben :'); read (n);
  main(n);
end.
```

Ablaufprotokoll:

```
Dimension n eingeben : 4

Matrix A eingeben:

1e9   8   126     -237
100   2   -12     1
1e5   10  -1e7    81
13    -3  30      1e-7

Matrix B eingeben

1e8   85     8      6
12    3      1e3    156
3     14     1e10   13
2     -8332  -1e4   -1e-8
```

a) mit 13-stelliger Dezimalarithmetik

```
Spur von A*B herkoemmlich berechnet  : -1.000000000000E-15
Spur von A*B maximal genau berechnet :  6.000000000000E+00
```

b) mit 53-stelliger Binärarithmetik

```
Spur von A*B herkoemmlich berechnet  : -9.999999999999999E-016
Spur von A*B maximal genau berechnet :  5.999999999999999E+000
```

Aufgabe 24: Taschenrechner für Polynome

Schreiben Sie ein PASCAL–XSC Programm, das einen Taschenrechner für die Addition und Multiplikation von Polynomen mit ganzzahligen Koeffizienten realisiert. Der Grad n der Eingabepolynome soll maximal 5 sein. Für zwei Polynome p und q vom Grad n mit

$$p(x) = \sum_{i=0}^{n} a_i x^i, \quad q(x) = \sum_{i=0}^{n} b_i x^i$$

ist die Summe s erklärt durch

$$s(x) = p(x) + q(x) = \sum_{i=0}^{n} (a_i + b_i) x^i$$

und das Produkt r durch

$$r(x) = p(x) \cdot q(x) = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j x^{i+j}.$$

Ein zu implementierendes Modul soll folgende Teile enthalten:

a) eine dynamische Typvereinbarung `POLYNOM`, worin ein Polynom als dynamisches *integer*-Feld festgelegt wird,

b) eine Prozedur für die Eingabe der Polynomkoeffizienten a_i bzw. b_i,

c) einen Operator + mit zwei Operanden vom Typ `POLYNOM`, dessen Ergebnis-Polynom den gleichen Grad wie die Operanden besitzt,

d) einen Operator $*$ mit zwei Operanden vom Typ `POLYNOM`, dessen Ergebnis-Polynom den doppelten Grad besitzt,

e) eine Prozedur für die Ausgabe der Polynome.

Ein Programm, das dieses Modul testet, soll eine Prozedur `MAIN` mit Parameter n (≤ 5) enthalten, in der drei Polynome (p, q, s) vom Grad n und ein Polynom (r) vom Grad $2n$ vereinbart, p und q eingelesen und abhängig von einer Benutzereingabe die Summe s bzw. das Produkt r berechnet und ausgegeben werden. Im Hauptprogramm dieses Testprogramms soll lediglich der Grad der Polynome eingegeben und die Prozedur `MAIN` aufgerufen werden.

Lösung:

```
module poly;

global type  polynom = dynamic array [*] of integer;

global procedure read (var f: text; var a:polynom);
  var i:integer;
  begin
    for i:= 0 to ubound(a) do
    begin
      read (f, a[i]);
    end;
  end;

global operator + (a,b: polynom) resp: polynom[0..ubound(a)];
  var i: integer;
  begin
    for i:= 0 to ubound(a) do
      resp[i]:= a[i] + b[i];
  end;

global operator * (a,b: polynom) resm: polynom[0..2*ubound(a)];
  var i,j: integer;
      r  : polynom[0..2*ubound(a)];
  begin
    for i:= 0 to 2*ubound(a) do
        r[i]:= 0;
    for i:= 0 to ubound(a) do
      for j:= 0 to ubound (a) do
        r[i+j]:= r[i+j] + a[i] * b[j];
    resm:= r;
  end;

global procedure write (f: text; a:polynom);
  var u,i:integer;
  begin
    u:= ubound(a);
    write(f, a[0]:1);
    if u>0 then
      write(f, ' + ',a[1]:1,'x');
    for i:= 2 to u do
      write(f, ' + ',a[i]:1,'x^',i:1);
  end;

end. {module poly}
```

```
program test_poly (input,output);

use poly;

var
  n, o : integer;

procedure main (n : integer; var o : integer);
  var
    p,q,s : polynom[0..n];
    r     : polynom[0..2*n];
  begin
      writeln('Koeffizienten von p eingeben:');
      read (p);
      writeln;

      writeln('Koeffizienten von q eingeben:');
      read (q);
      writeln;

      repeat
        writeln('Bitte waehlen:');
        writeln(' p + q               ==> 0');
        writeln(' p * q               ==> 1');
        writeln(' neue Polynome p,q   ==> 2');
        writeln(' Ende des Programms  ==> 9');
        writeln;
        write  ('Auswahl:   ==> '); read(o);
        writeln;

        if o = 0 then
          begin
            s:= p+q;
            write('p   = '); writeln(p);
            write('q   = '); writeln(q);
            write('p+q = '); writeln(s);
          end
        else if o = 1 then
          begin
            r:= p*q;
            write('p   = '); writeln(p);
            write('q   = '); writeln(q);
            write('p*q = '); writeln(r);
          end;
      until (o<>0) and (o<>1);
      writeln;
  end;
```

```
begin
  writeln('*** Taschenrechner fuer Polynome ***'); writeln;
  repeat
    repeat
      write('Grad der Polynome (>= 0 und <= 5) : '); read (n);
    until (0<=n) and (n<=5);
    writeln;
    main (n,o);
  until (o=9);
end.
```

Ablaufprotokoll:

```
*** Taschenrechner fuer Polynome ***

Grad der Polynome (>= 0 und >=5) : 4

Koeffizienten von p eingeben:
99  11  22  33  44

Koeffizienten von q eingeben:
0  1  2  3  4

Bitte waehlen:
 p + q               ==> 0
 p * q               ==> 1
 neue Polynome p,q   ==> 2
 Ende des Programms  ==> 9

Auswahl:   ==> 0

p   = 99 + 11x + 22x^2 + 33x^3 + 44x^4
q   =  0 +  1x +  2x^2 +  3x^3 +  4x^4
p+q = 99 + 12x + 24x^2 + 36x^3 + 48x^4

Bitte waehlen:
 p + q               ==> 0
 p * q               ==> 1
 neue Polynome p,q   ==> 2
 Ende des Programms  ==> 9

Auswahl:   ==> 1

p   = 99 +  11x   + 22x^2  + 33x^3  + 44x^4
q   =  0 +   1x   +  2x^2  +  3x^3  +  4x^4
p*q =  0 +  99x   + 209x^2 + 341x^3 + 506x^4
         + 220x^5 + 275x^6 + 264x^7 + 176x^8
```

Aufgabe 25: Intervall-Newton-Verfahren

Die Intervall-Einschließung X_n einer Nullstelle einer Funktion $f(x)$, deren Ableitung auf dem Intervall $[a, b]$ stetig und ungleich null ist, kann mit Hilfe des Intervall-Newton-Verfahrens

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

verbessert werden. Dabei ist $m(X)$ der Mittelpunkt von X.

Schreiben Sie unter Verwendung des Moduls I_ARI ein PASCAL–XSC Programm, das mit diesem Verfahren eine Einschließung der Nullstelle von

$$f(x) = \sqrt{x} + (x+1)\cos x$$

berechnet. Entwerfen Sie dazu

- eine Funktion `F`, die $f(X)$ intervallmäßig berechnet,
- eine Funktion `DF`, die die Ableitung $f'(x)$ intervallmäßig berechnet,
- eine Funktion `M`, die den Mittelpunkt m des Intervalles $X = [x_1, x_2]$ unter Verwendung eines #-Ausdrucks maximal genau berechnet,
- ein Hauptprogramm, das das Startintervall $X = [a, b]$ einliest, die zwei Kriterien $f(a) \cdot f(b) < 0$ und $0 \notin$ `DF`(X) überprüft und die Iterationen nach dem Newton-Verfahren durchführt. Dabei soll in jedem Schritt das neu berechnete Intervall ausgegeben werden. Die Iteration soll abgebrochen werden, wenn gilt: $X_{n+1} = X_n$.

Hinweis: Verwenden Sie $[2.0, 3.0]$ als Startintervall für die Iteration.

Beachten Sie, daß zur Berechnung von $f(m(X))$ mit der Intervallfunktion `F` der durch `M` berechnete Mittelpunkt erst wieder in ein (Punkt-)Intervall gewandelt werden muß.

Lösung:

```
program i_newton (input,output);

use i_ari;

var  X,Y : interval;

function F (X : interval) : interval;
  begin
    F:= sqrt(X) + (X+1) * cos(X);
  end;
```

```
function DF (X : interval) : interval;
  begin
    DF:= 0.5/sqrt(X) + cos(X) - (X + 1) * sin(X)
  end;

function m (X : interval) : real;
  begin
    m:= #* (0.5 * X.inf + 0.5 * X.sup);
  end;

function Krit (X : interval) : boolean;
  var
    A, B: interval;
  begin
    A   := intval(inf(X));
    B   := intval(sup(X));
    Krit:= (sup(F(A)*F(B)) < 0) and (not (0 in DF(X)));
  end;

begin
  write ('Startintervall: '); read (Y);
  writeln;
  writeln ('Iteration'); writeln;
  if Krit(Y) then
    repeat
      writeln (Y);
      X:= Y;
      Y:= ( m(X) - F ( intval(m(X)) ) / DF(X) ) ** X;
    until Y = X
  else
    writeln ('Kriterium nicht erfuellt');
end.
```

Ablaufprotokoll:

```
Startintervall: [2.0,3.0]

Iteration

[             2.0E+00,             3.0E+00]
[             2.0E+00,             2.3E+00]
[            2.05E+00,            2.07E+00]
[         2.05903E+00,         2.05906E+00]
[  2.059045253413E+00,  2.059045253417E+00]
[  2.059045253415E+00,  2.059045253416E+00]
```

Aufgabe 26: Runge-Kutta-Verfahren

Das Runge-Kutta-Verfahren wird benutzt zur approximativen Lösung von Anfangswertaufgaben der Form

$$Y' = F(x, Y); \quad Y(x^0) = Y^0;$$

mit

$$Y = \begin{pmatrix} y_1(x) \\ \vdots \\ y_n(x) \end{pmatrix}, \; Y' = \begin{pmatrix} y_1'(x) \\ \vdots \\ y_n'(x) \end{pmatrix}$$

und

$$F(x, Y) = \begin{pmatrix} f_1(x, y_1, \dots, y_n) \\ \vdots \\ f_n(x, y_1, \dots, y_n) \end{pmatrix}.$$

Eine Näherung der Lösung Y an der Stelle $x + h$ ist mit den durch

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

definierten Koeffizienten K_i gegeben durch die Formel

$$Y(x + h) = Y(x) + (K_1 + 2K_2 + 2K_3 + K_4)/6.$$

Schreiben Sie ein PASCAL–XSC Programm, das unter Verwendung des Moduls MV_ARI ausgehend von $Y(0) = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ die Werte von Y an den Stellen $x_i = i * h$, $i = 1, \dots, 10$ mit $h = 0.1$ für die Funktion

$$F(x, Y) = \begin{pmatrix} Y_1 - Y_2 \\ e^x \cdot Y_3 \\ (Y_1 - Y_2)/e^x \end{pmatrix}$$

berechnet und ausgibt. Definieren Sie dazu die Vektorfunktion `F(x,Y)` und berechnen Sie dann in einer Schleife unter Verwendung der in MV_ARI vordefinierten Operatoren jeweils die Ausdrücke `K1`, `K2`, `K3`, `K4` und den Wert von $Y(x_i)$.

Lösung:

```
program Runge (input,output);

use mv_ari;

const
  n = 3;
var
  h, x, xi             : real;
  Y, k1, k2, k3, k4    : rvector[1..n];
  i                    : integer;

function F (x : real; Y : rvector) : rvector[1..n];
  var
    i: integer;
  begin
    F[1]:= Y[1] - Y[2];
    F[2]:= exp(x) * Y[3];
    F[3]:= (Y[1] - Y[2]) / exp(x);
  end;

begin
  xi:= 0;  Y[1]:= 1;  Y[2]:= 0;  Y[3]:= 1;  h:= 0.1;
  writeln ('       x                                   Y');
  write   ('-------------------------------------');
  writeln ('-------------------------------------');
  writeln (xi:7:4,'     ',Y[1],'  ',Y[2],'  ',Y[3]);
  for i:=1 to 10 do
  begin
    x  := xi;
    xi := i*h;

    k1 := h * F (x      , Y);
    k2 := h * F (x + h/2, Y + k1/2);
    k3 := h * F (x + h/2, Y + k2/2);
    k4 := h * F (x + h  , Y + k3);

    Y  := Y + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    writeln (xi:7:4,'     ',Y[1],'  ',Y[2],'  ',Y[3]);
  end;
end.
```

Ablaufprotokoll:

```
  x                               Y
-----------------------------------------------------------------------
 0.0000     1.000000000000E+00  0.000000000000E+00  1.000000000000E+00
 0.1000     1.099649504200E+00  1.103327288503E-01  1.094837415056E+00
 0.2000     1.197055668877E+00  2.426546848268E-01  1.178735575069E+00
 0.3000     1.289568804504E+00  3.989095750025E-01  1.250856198787E+00
 0.4000     1.374060725876E+00  5.809424489455E-01  1.310478682209E+00
 0.5000     1.446887955451E+00  7.904370738033E-01  1.357007298606E+00
 0.6000     1.503858169531E+00  1.028843008954E+00  1.389977150813E+00
 0.7000     1.540201346905E+00  1.297291711580E+00  1.409058816313E+00
 0.8000     1.550547298813E+00  1.596501098943E+00  1.414061638705E+00
 0.9000     1.528911489731E+00  1.926668121123E+00  1.404935632665E+00
 1.0000     1.468691297995E+00  2.287349064239E+00  1.381771983374E+00
```

Bemerkung: In den Lösungen zu den beiden letzten Aufgaben wird deutlich, daß das allgemeine Operatorkonzept in PASCAL–XSC die Übertragung von numerischen Algorithmen in Programmcode wesentlich vereinfacht. Im Prinzip lassen sich die mathematischen Formeln direkt als Anweisungszeilen übernehmen.

Aufgabe 27: Rationalarithmetik

Entwerfen Sie ein PASCAL–XSC Modul für eine Rationalarithmetik. Eine rationale Zahl $p = z/n$ soll dabei dargestellt werden als ein *record*-Typ mit den Komponenten z für den Zähler und n für den Nenner von p (z, n *integer*-Werte mit $n > 0$). Im Modul sollen global zur Verfügung gestellt werden

1) der Typ `RATIONAL`,

2) die Operatoren `+,-,*,/`, die als Ergebnis einen *gekürzten* Bruch vom Typ `RATIONAL` liefern,

3) je eine Prozedur zur Eingabe bzw. zur Ausgabe von Rationalzahlen in der Form *integer*/*integer*.

Für die Implementierung der Operatoren werden Funktionen für das Kürzen und für die Berechnung des größten gemeinsamen Teilers (ggT) benötigt, die nur für die lokale Verwendung im Modul vereinbart sein sollen.

Ein Testprogramm soll die einzelnen Operatoren testen und den Wert des Ausdrucks

$$(a + b) * (b - c)/(c + d)$$

für $a = 3/4$, $b = 2/7$, $c = 4/5$ und $d = 7/9$ berechnen.

Hinweis: Die Funktion für das Kürzen soll durch die ganzzahlige Division (**div**) von Zähler und Nenner durch den ggT realisiert werden. Die Bestimmung des ggT's soll durch den folgenden Algorithmus erfolgen:

$$a, b > 0; \quad z_0 := a; \quad n_0 := b; \quad i := 0;$$

$$\text{Setze } \left\{ \begin{array}{lcl} z_{i+1} & := & n_i \\ n_{i+1} & := & z_i \ \mathbf{mod} \ n_i \end{array} \right\} \text{ für } i = 0, 1, 2, \ldots$$

solange, bis gilt $n_{i+1} = 0$.

Dann ist z_{i+1} (bzw. n_i) der ggT von a und b.

Zu beachten ist: $\text{ggT}(0, x) = x$ für jedes $x \neq 0$.

Lösung:

```
module ratio;

global type
  positive = 1..maxint;

global type
  rational = record
               z : integer;
               n : positive;
             end;

function ggt (a, b : integer) : positive;
  var
    z, n, r : integer;
  begin
    if a = 0 then
      ggt:= b
    else if b = 0 then
      ggt:= a
    else
      begin
        n:= a;
        r:= b;
        repeat
          z:= n;
          n:= r;
          r:= z mod n;
        until r = 0;
        ggt:= abs(n);
      end;
  end;

function kuerze (a: rational) : rational;
  var
    g : positive;
  begin
    g:= ggt (abs(a.z),a.n);
    if (g = 0) or (g = 1) then
      kuerze := a
    else
      begin
        kuerze.z:= a.z div g;
        kuerze.n:= a.n div g;
      end;
  end;
```

```
global operator + (a,b : rational) respl : rational;
  var
    s : rational;
  begin
    s.z := a.z*b.n + b.z*a.n;
    s.n := a.n*b.n;
    respl:= kuerze (s);
  end;

global operator - (a,b : rational) resmi : rational;
  var
    s : rational;
  begin
    s.z := a.z*b.n - b.z*a.n;
    s.n := a.n*b.n;
    resmi:= kuerze (s);
  end;

global operator * (a,b : rational) resmu : rational;
  var
    s : rational;
  begin
    s.z := a.z*b.z;
    s.n := a.n*b.n;
    resmu:= kuerze (s);
  end;

global operator / (a,b : rational) resdi : rational;
  var
    help : integer;
    s    : rational;

  begin
    s.z := a.z*b.n;
    help:= a.n*b.z;
    if help >= 0 then
      s.n:= help * help div help {eventuell Division durch Null}
    else
      begin
        s.z := -s.z;
        s.n := -help;
      end;
    resdi:= kuerze (s);
  end;
```

```
global procedure read (var f: text; var r: rational);
  var
    s, sz, sn: string;
    i, l: integer;
  begin
    readln(f);
    read (f,s);
    i:= pos ('/',s);
    l:= length(s);
    sz:= substring(s,1,i-1);
    sn:= substring(s,i+1,l-i);
    r.z:= ival(sz);
    l  := ival(sn);
    if l >= 0 then
      r.n:= l * l div l  {eventuell Division durch Null}
    else
      begin
        r.z := - r.z;
        r.n := - l;
      end;
  end;

global procedure write (var f: text; a: rational);
  begin
    write (f,a.z:1,'/',a.n:1);
  end;

begin end.
```

```
program test_ratio (input,output);

use ratio;

var
  a,b,c,d : rational;

begin
  write ('a = '); read (a);
  writeln (a);
  write ('b = '); read (b);
  writeln (b);
  write ('c = '); read (c);
  writeln (c);
  write ('d = '); read (d);
  writeln (d);
  writeln ('a+b = ', a+b);
  writeln ('b-c = ', b-c);
  writeln ('c+d = ', c+d);
  writeln ('(a+b)*(b-c)/(c+d) = ', (a+b)*(b-c)/(c+d));
end.
```

Ablaufprotokoll:

```
a   =    3/4
b   =    2/7
c   =    4/5
d   =    7/9

a+b =  29/28
b-c = -18/35
c+d =  71/45
(a+b)*(b-c)/(c+d) = -2349/6958
```

Aufgabe 28: Polynomauswertung

Mit Hilfe des Moduls LSS aus der PASCAL–XSC Numerikbibliothek zur verifizierenden Lösung linearer Gleichungssysteme soll die Auswertung eines Polynoms

$$p(t) = a_n t^n + \cdots + a_1 t + a_0$$

mit maximaler Genauigkeit erfolgen. Ausgehend vom Hornerschema

$$p(t) = (\ldots(a_n \cdot t + a_{n-1}) \cdot t + a_{n-2}) \cdots) \cdot t + a_1) \cdot t + a_0$$

läßt sich die Auswertung des Polynoms durch Einführen der $n+1$ Variablen $x_0, x_1, \ldots, x_{n-1}, x_n$ auf das Lösen des linearen Gleichungssystems

$$\begin{aligned} x_0 &= a_n \\ x_1 &= t x_0 + a_{n-1} \\ &\vdots \\ x_{n-1} &= t x_{n-2} + a_1 \\ x_n &= t x_{n-1} + a_0 \end{aligned}$$

zurückführen, wobei der Wert des Polynoms p an der Stelle t durch x_n gegeben ist, d. h. $x_n = p(t)$.

Zu lösen ist also das lineare Gleichungssystem

$$\begin{pmatrix} 1 & & & & 0 \\ -t & 1 & & & \\ & \ddots & \ddots & & \\ & & -t & 1 & \\ 0 & & & -t & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

bzw.

$$Ax = b$$

mit

$$A = (a_{ij}), \quad a_{ij} = \left\{ \begin{array}{rl} 1 & \text{für } i = j \\ -t & \text{für } i = j+1 \\ 0 & \text{sonst} \end{array} \right\}, \quad i, j = 0, \ldots, n$$

und

$$b = (b_i), \quad b_i = a_{n-i}, \quad i = 0, \ldots, n.$$

Schreiben Sie ein PASCAL–XSC-Programm, das folgende Teile enthält:

(a) eine dynamische Typvereinbarung *polynom* (Komponententyp *real*),

(b) eine Prozedur zum Einlesen von Polynomen (Polynomkoeffizienten),

(c) eine Funktion *horner* zur Berechnung des Polynomwertes mit dem Hornerschema,

(d) eine Prozedur *set_A_b*, die aus einem Polynom p und einer *real*-Zahl t die Matrix A und den Vektor b erzeugt,

(e) eine Prozedur *main* mit formalem Parameter n, in der

- eine Variable p vom Typ *polynom*, ein Vektor b vom Typ *rvector*, ein Intervallvektor X vom Typ *ivector* und eine quadratische Matrix A vom Typ *rmatrix* jeweils mit Indexbereich $0, \ldots, n$ vereinbart werden,
- mit Hilfe von Teil (b) die Polynomkoeffizienten $a_0, \ldots, a_n$ von p und die Auswertestelle t eingelesen werden,
- mit Hilfe von Teil (d) die Matrix A und der Vektor b erzeugt werden,
- mittels der Prozedur *lss* die maximal genaue Einschließung X der Lösung des Gleichungssystems $Ax = b$ berechnet wird
- und abschließend, falls *lss* fehlerfrei ausgeführt wurde, die linke und rechte Grenze der Einschließung X_n des Polynomwertes $x_n = p(t)$ und der mit dem Hornerschema (Teil (c)) berechnete Wert zum Vergleich ausgegeben werden,

(f) ein Hauptprogramm, in dem der Polynomgrad n eingelesen und die Prozedur *main* aufgerufen wird.

Hinweis: Verwenden Sie das Modul LSS aus der PASCAL–XSC-Numerikbibliothek. Die in diesem Modul zur Verfügung gestellte Prozedur *lss* liefert zur Matrix A und der rechten Seite b einen verifiziert berechneten Einschließungsvektor X der Lösung x von $Ax = b$. Die Schnittstelle dieser Prozedur sieht folgendermaßen aus:

```
procedure lss (  var A: rmatrix;   var b: rvector;
                 var X: ivector;   var errcode: integer  );
```

Dabei bedeutet:

errcode = 0 : Ausführung fehlerfrei,
errcode = 1 : System zu schlecht konditioniert,
errcode = 2 : Matrix möglicherweise singulär.

Testen Sie Ihr Programm an folgenden Beispielen:

Beispiel 1:

Polynomgrad	3
Koeffizienten	$a_0 = 1536796802$
	$a_1 = -1086679440$
	$a_2 = -768398401$
	$a_3 = 543339720$
Auswertestelle	$t = 1.4142135$

Beispiel 2:

Polynomgrad	3
Koeffizienten	$a_0 = 170.4$
	$a_1 = -356.41$
	$a_2 = 168.97$
	$a_3 = 18.601$
Auswertestelle	$t = 0.916079759$

Lösung:

```
program poly_wert (input,output);

use i_ari, lss;

type
  polynom = dynamic array [*] of real;


procedure read (var f: text; var p: polynom);
  var
    i: integer;
  begin
    for i:= 0 to ubound(p) do
    begin
      write ('Koeff. ',i:2,': ');
      read (f, p[i]);
    end;
  end;


function horner (p : polynom; t: real) : real;
  var
    h: real;
    i: integer;
  begin
    h:= 0;
    for i:= ubound(p) downto 0 do
      h:= p[i] + t * h;
    horner:= h;
  end;
```

```
procedure set_A_b (p: polynom; t: real; var A: rmatrix; var b: rvector);
  var
    i,j: integer;
  begin
    for i:= lbound(A,1) to ubound(A,1) do
      for j:= lbound(A,2) to ubound(A,2) do
        if i=j then
          A[i,j] := 1
        else if i=j+1 then
          A[i,j] := -t
        else
          A[i,j] := 0;
    for i:= lbound(b) to ubound(b) do
      b[i] := p[ubound(p)-i];
  end;

procedure main (n: integer);
  var
    p: polynom[0..n];
    b: rvector[0..n];
    X: ivector[0..n];
    A: rmatrix[0..n,0..n];
    t: real;
    error: integer;
  begin
    writeln ('Polynom eingeben');
    read (p);
    write ('Auswertestelle  t = '); read(t);
    writeln;
    set_A_b (p,t,A,b);
    lss (A,b,X,error);
    if error=0 then
    begin
      writeln('Hornerschema  :  ', horner (p,t));
      write  ('Einschliessung: '); writeln (X[n]);
    end
    else
      writeln ('Error ',error:1,' aufgetreten');
  end;

var  n: integer;

begin
  write ('Polynomgrad: ');  read (n);
  main (n);
end.
```

Ablaufprotokoll:

Beispiel 1

```
Polynomgrad: 3
Polynom eingeben
Koeff.  0 =  1536796802
Koeff.  1 = -1086679440
Koeff.  2 = -768398401
Koeff.  3 =  543339720
Auswertestelle  t = 1.4142135
```

a) mit 13-stelliger Dezimalarithmetik

```
Hornerschema  :   1.000000000000E-03
Einschliessung: [ 5.978758735594E-06, 5.978758735596E-06]
```

b) mit 53-stelliger Binärarithmetik

```
Hornerschema  :   5.96046447753906E-006
Einschliessung: [ 5.97875873324932E-006, 5.97875873324934E-006]
```

Beispiel 2

```
Polynomgrad: 3
Koeff.  0 =  170.4
Koeff.  1 = -356.41
Koeff.  2 =  168.97
Koeff.  3 =   18.601
Auswertestelle  t = 0.916079759
```

a) mit 13-stelliger Dezimalarithmetik

```
Hornerschema  :   0.000000000000E+00
Einschliessung: [ 1.148703133435E-13, 1.148703133436E-13]
```

b) mit 53-stelliger Binärarithmetik

```
Hornerschema  :   8.52651282912120E-014
Einschliessung: [ 9.59892282082942E-014, 9.59892282082943E-014]
```

Bemerkung: Diese letzte Aufgabe demonstriert die Anwendung des verifizierenden Gleichungssystemlösers zur maximal genauen Berechnung von Polynomwerten. Die verifizierten Ergebnisse zeigen, daß das oft verwendete Hornerschema völlig unbrauchbare Ergebnisse liefern kann.

Der Einfachheit halber wurde hier die Prozedur *lss* verwendet. Aus Effektivitätsgründen wäre es günstiger, eine Rückwärtstransformation für die Lösung des Gleichungssystems durchzuführen (vgl. etwa [7]).

Auch hier sind die Unterschiede zwischen den mit Dezimal- und Binärarithmetik berechneten Werten auf die Konvertierungsfehler, die im Falle der Binärarithmetik bei der Eingabe unvermeidbar entstehen, zurückzuführen.

Anhang A

Syntaxdiagramme

In Ergänzung zu der in der Sprachbeschreibung verwendeten Syntaxbeschreibung in vereinfachter Backus-Naur-Form geben wir im folgenden die vollständige Syntax der Sprache PASCAL–XSC in Form von Syntaxdiagrammen an, wie sie z. B. in [6] oder in [12] verwendet werden. Die nachfolgenden Diagramme sind dabei wie folgt gestaltet.

- Jedes Diagramm ist unmittelbar hinter seiner Nummer mit einem Bezeichner (Folge von Großbuchstaben) gekennzeichnet. Dieser Bezeichner, auch Syntaxvariable genannt, ist jeweils so gewählt, daß er einen Hinweis auf das durch das Diagramm dargestellte Sprachelement gibt.
- Ein Diagramm ist aufgebaut aus Syntaxvariablen, Terminalsymbolen (fettgedruckte Wortsymbole und in Kreise bzw. Ovale eingeschlossene Symbole bzw. Folgen von Symbolen des Alphabets) und verbindende Linien (durchgezogen oder gepunktet).
- Eine Syntaxvariable kann innerhalb eines Diagramms auch mit einem semantischen Zusatz auftreten. So taucht z. B. die Variable NAME auch mit dem Zusatz *KOMP* auf, der darauf hindeutet, daß an dieser Stelle nur ein Komponentenname stehen darf. Für die Definition der Variablen *KOMP* NAME ist aber trotzdem das Syntaxdiagramm mit dem Bezeichner NAME gültig.

  Die Semantikzusätze werden außerdem auch durch *kursiv* geschriebene Erläuterungen gegeben, die meist direkt unter bzw. neben der betreffenden Syntaxvariablen zu finden sind. Dabei werden die folgenden Abkürzungen verwendet:

A	*Array*
B	*boolean*
CH	*char*
CD	*Code Typ, Aufzählungstyp*
CIR	*cinterval* (komplexe Intervalle)
CR	*complex* (komplexe Zahlen)
DOT	*dotprecision*

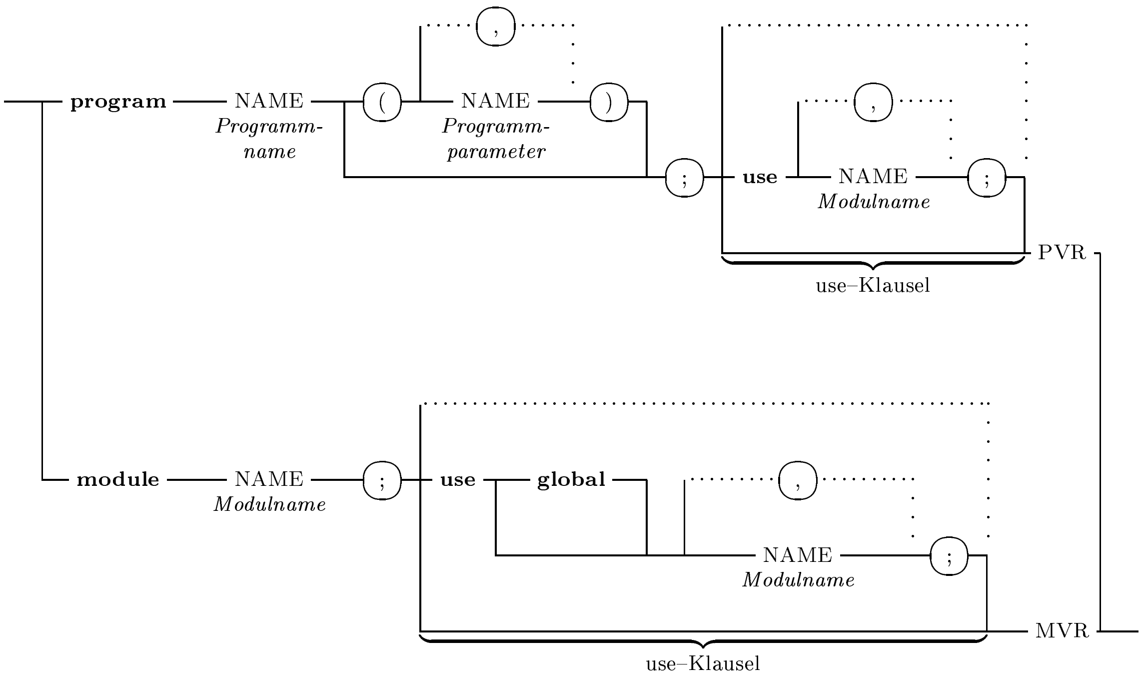
DSL	*Datensatzliste*
Dyadop	*Dyadischer Operator*
DYNA	*Dynamisches Array*
F	*File*
FKT	*Funktion*
FS	*Filestrukturtyp*
I	*integer*
IR	*interval* (reelle Intervalle)
KOMP	*Komponente (eines Records)*
KONST	*Konstante*
MCIR	*cimatrix* (komplexe Intervallmatrizen)
MCR	*cmatrix* (komplexe Matrizen)
MIR	*imatrix* (Intervallmatrizen)
MR	*rmatrix* (reelle Matrizen)
Monop	*Monadischer Operator*
P	*pointer*
R	*real*
REC	*Record*
RES	*Resultat*
ST	*string*
TF	*Textfile*
VCIR	*civector* (komplexe Intervallvektoren)
VCR	*cvector* (komplexe Vektoren)
VIR	*ivector* (Intervallvektoren)
VR	*rvector* (reelle Vektoren)
VAR	*Variable*

Im Anhang B.1 wird eine Liste aller vorkommenden Syntaxvariablen (Bezeichner) in alphabetischer Anordnung gegeben, die das Auffinden der zugehörigen Diagramme erleichtert.

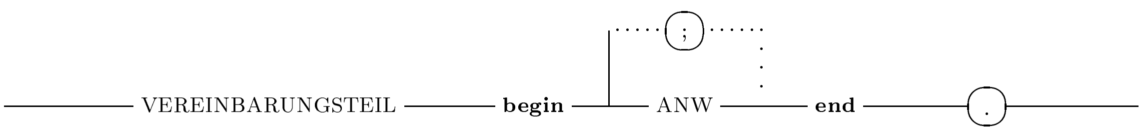
Die Verwendung der Syntaxdiagramme beim Schreiben von Programmen geschieht in folgender Weise:

- Der Durchlauf eines Diagramms beginnt jeweils am linken oberen Eingang.
- Durchgezogene Linien werden von links nach rechts bzw. von oben nach unten, gepunktete Linien in umgekehrter Richtung durchlaufen.
- Der Durchlauf eines Diagramms endet am rechten unteren Ausgang.
- Tritt beim Durchlauf eines Diagramms eine Syntaxvariable auf, so ist an dieser Stelle der aktuelle Durchlauf zu unterbrechen, das entsprechende, mit dieser Syntaxvariablen bezeichnete, Diagramm zu durchlaufen und anschließend der Durchlauf des ursprünglichen Syntaxdiagramms fortzusetzen.

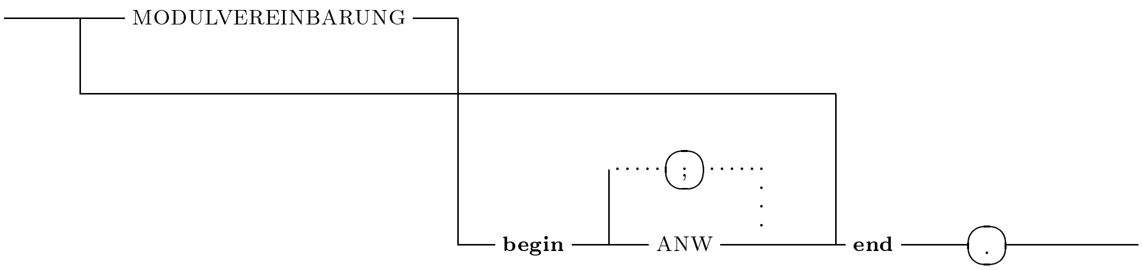
P1 ÜBERSETZUNGSEINHEIT



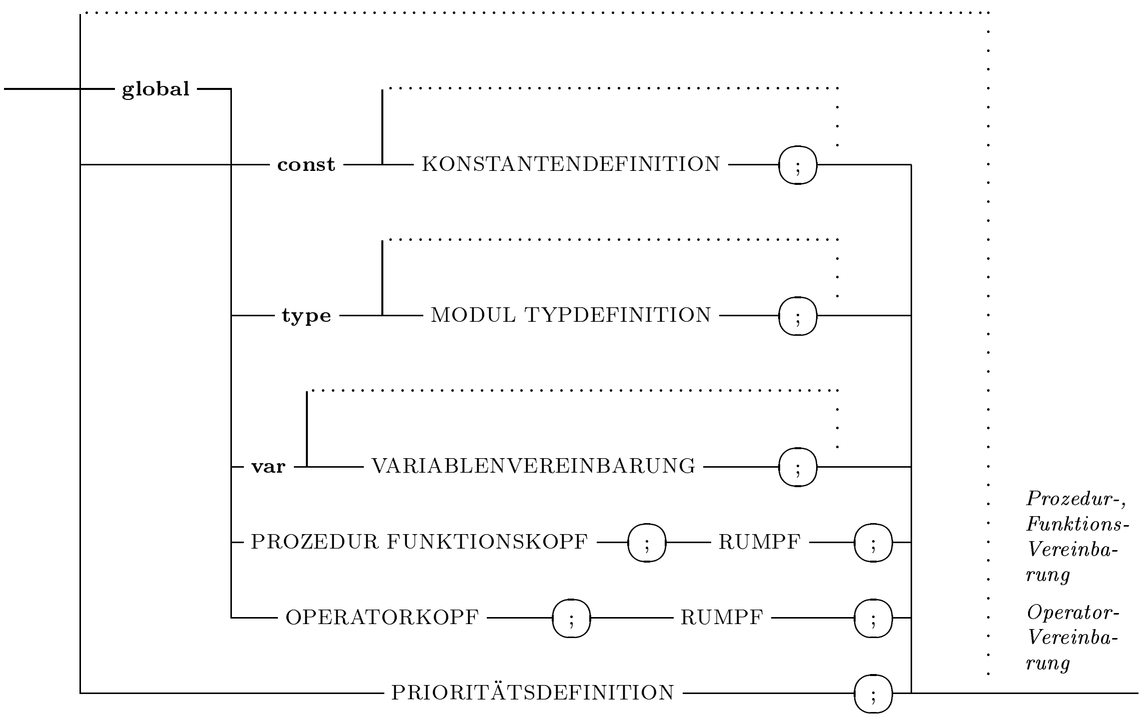
P2 PROGRAMM VEREINBARUNG UND RUMPF (PVR)



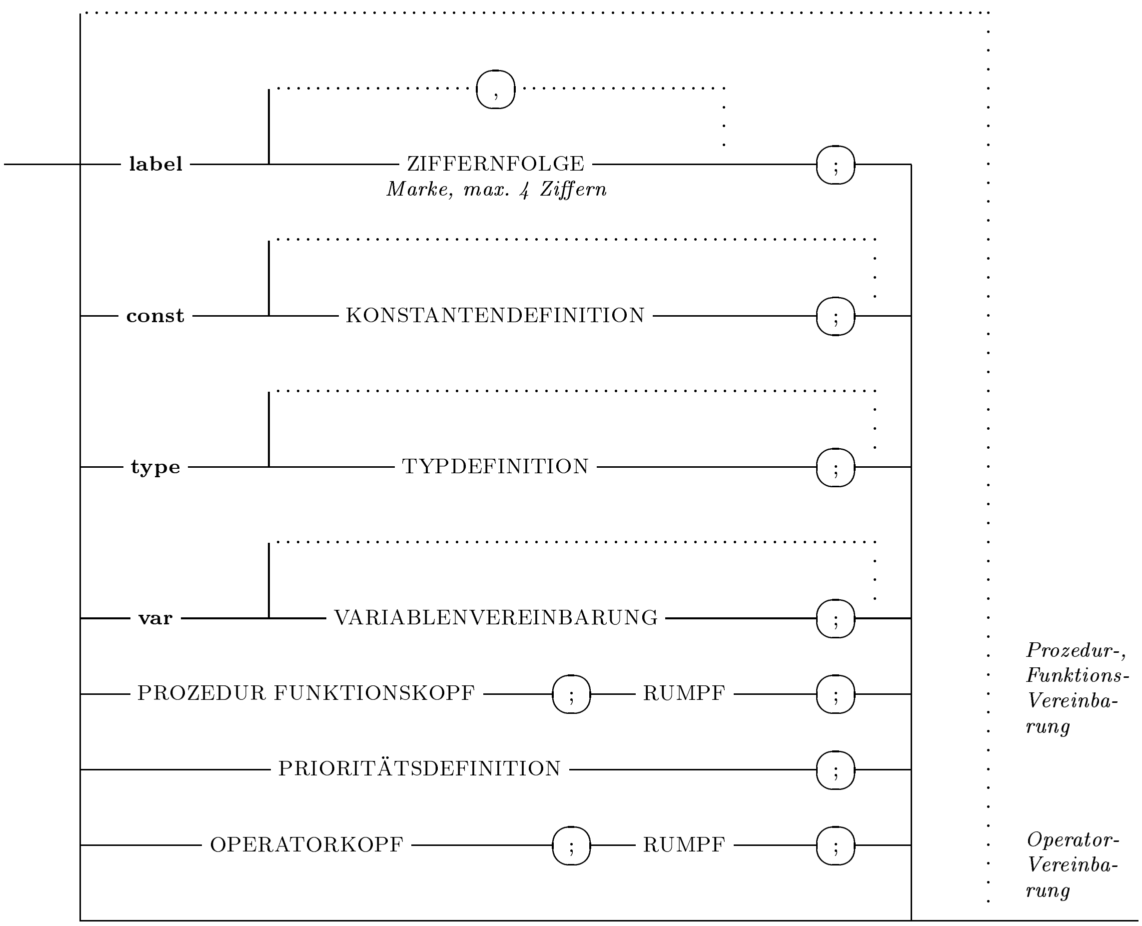
P3 MODUL VEREINBARUNG UND RUMPF (MVR)



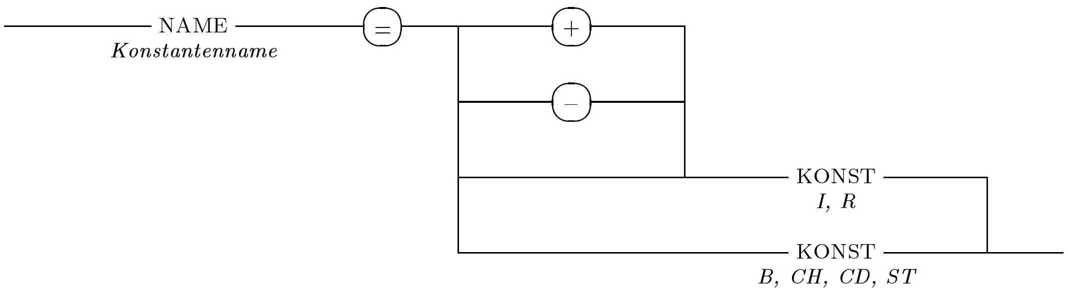
P4 MODULVEREINBARUNG



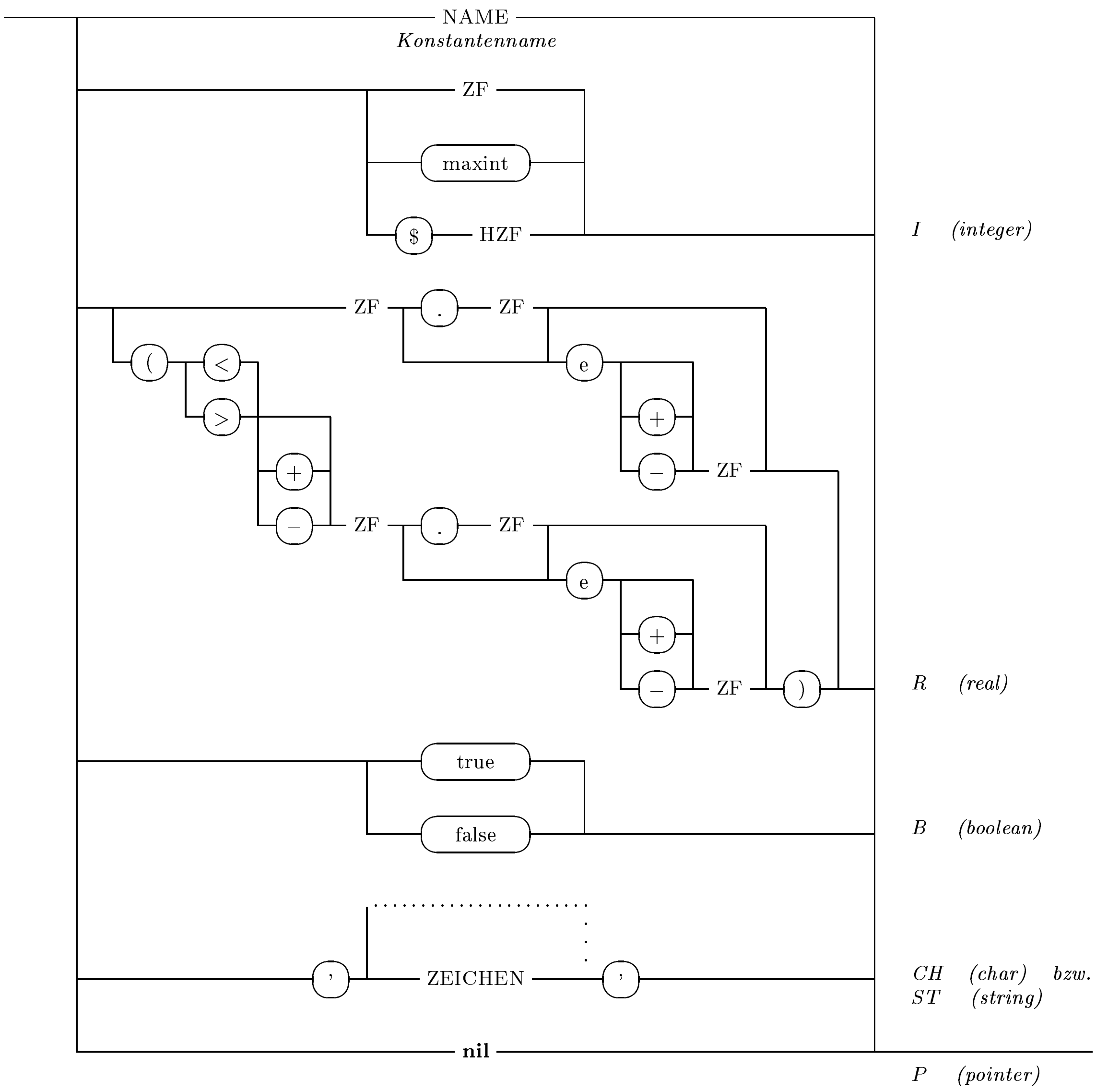
P5 VEREINBARUNGSTEIL



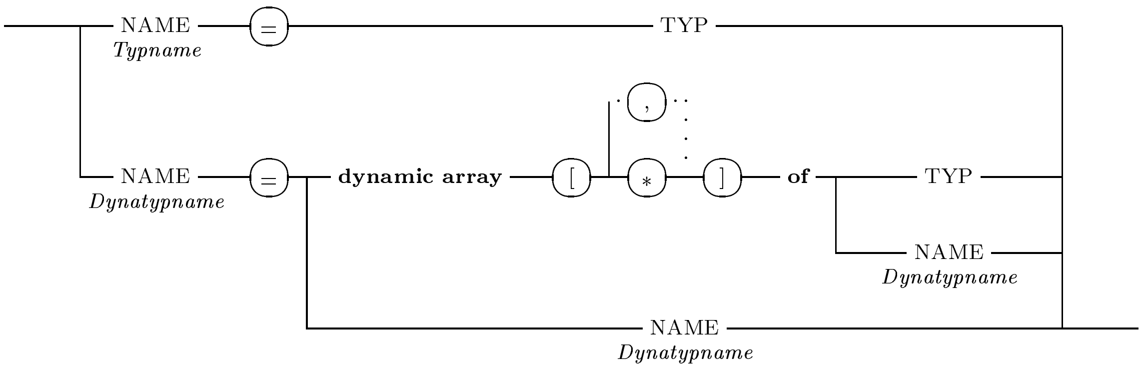
P6 KONSTANTENDEFINITION



P7 KONSTANTE (KONST)



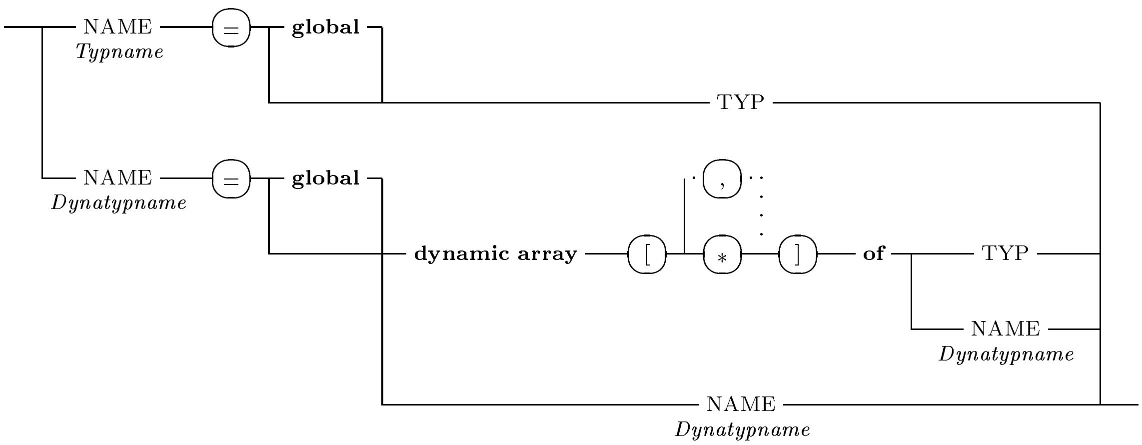
P8 TYPDEFINITION



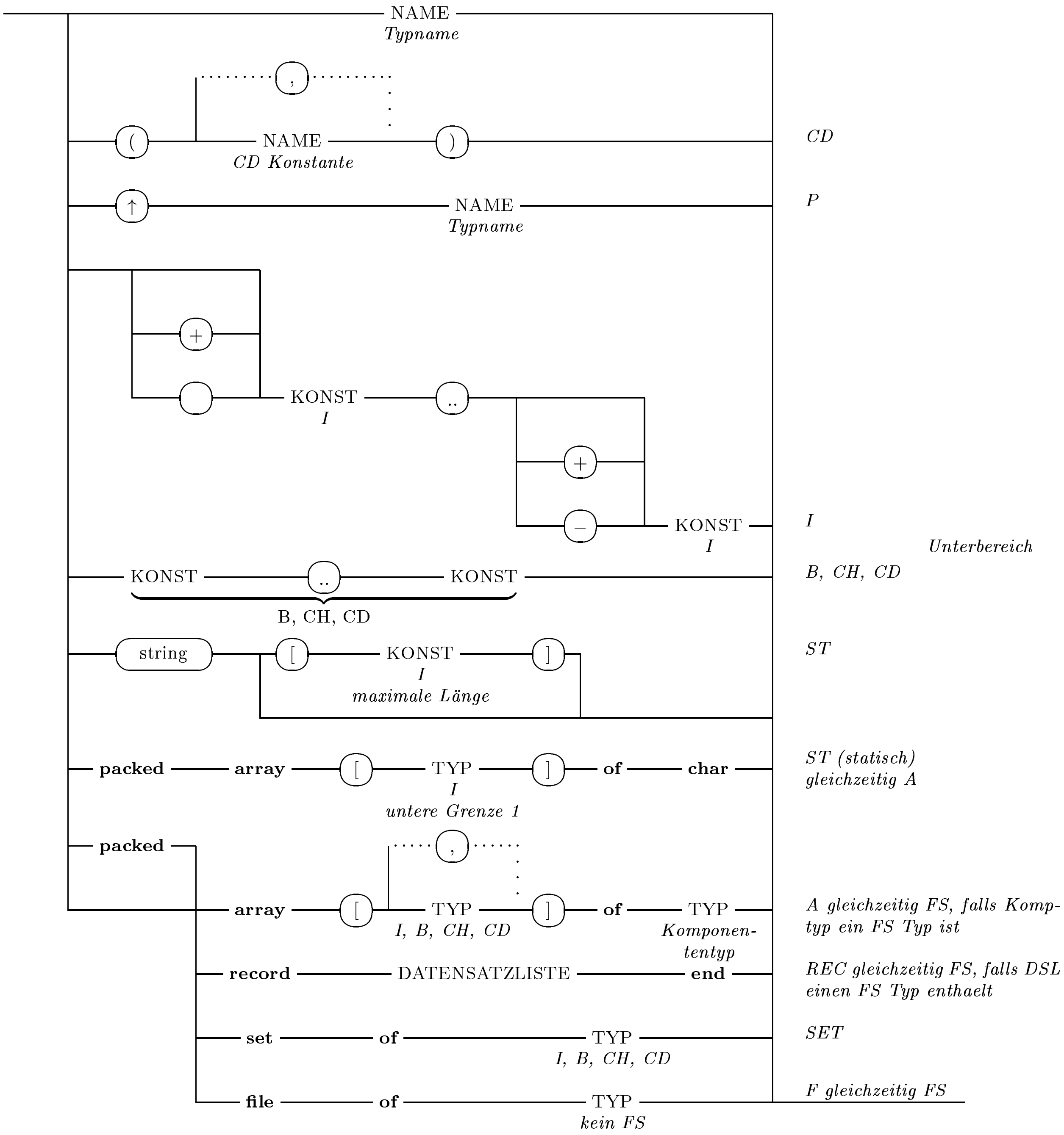
Liste der vordefinierten PASCAL–XSC Typnamen:

integer	I	dotprecision	DOT
real	R	rvector	VR
boolean	B	rmatrix	MR
char	CH	cvector	VCR
text	TF	cmatrix	MCR
string	ST	ivector	VIR
complex	CR	imatrix	MIR
interval	IR	civector	VCIR
cinterval	CIR	cimatrix	MCIR

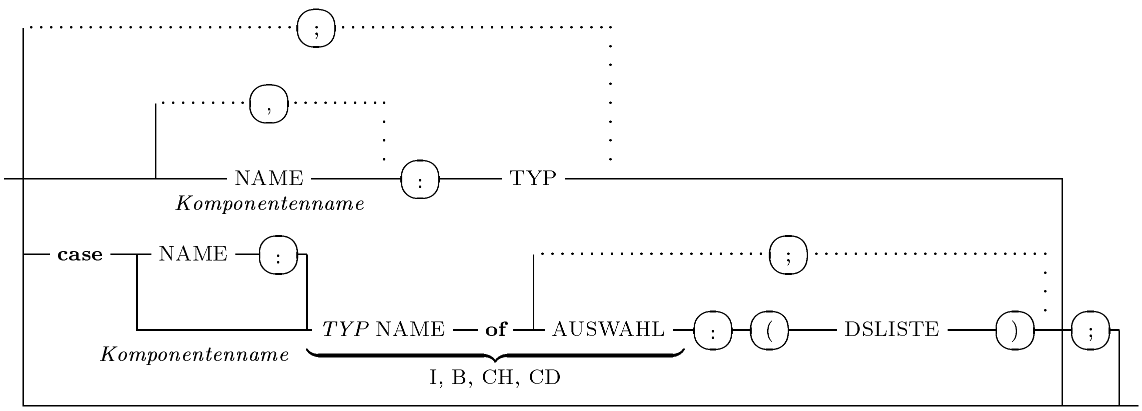
P9 MODUL TYPDEFINITION



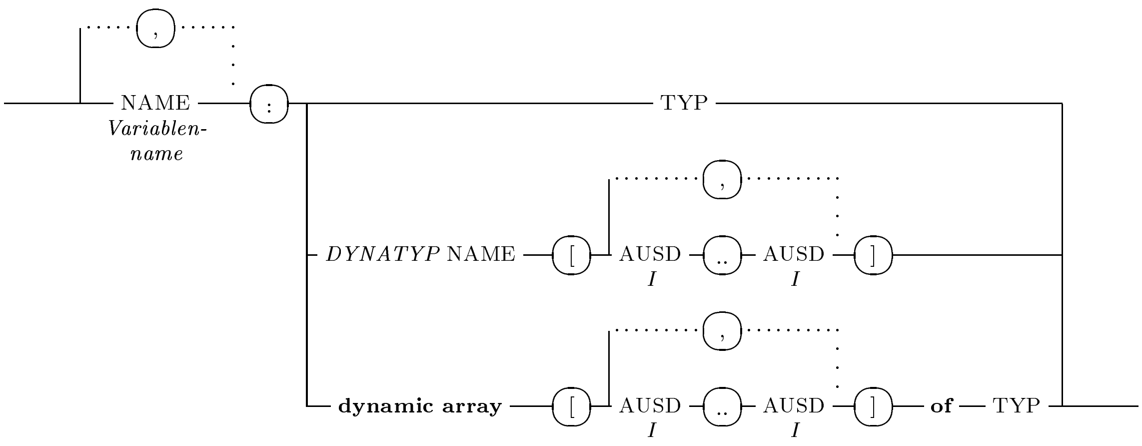
P10 TYP


NAME
Typname
(
,
NAME
CD Konstante
)
CD
↑
NAME
Typname
P
+
–
KONST
I
..
+
–
KONST
I
I
Unterbereich
KONST
..
KONST
B, CH, CD
B, CH, CD
string
[
KONST
I
maximale Länge
]
ST
packed
array
[
TYP
I
untere Grenze 1
]
of
char
ST (statisch)
gleichzeitig A
packed
,
array
[
TYP
I, B, CH, CD
]
of
TYP
Komponen-
tentyp
A gleichzeitig FS, falls Komp-
typ ein FS Typ ist
record
DATENSATZLISTE
end
REC gleichzeitig FS, falls DSL
einen FS Typ enthaelt
set
of
TYP
I, B, CH, CD
SET
file
of
TYP
kein FS
F gleichzeitig FS

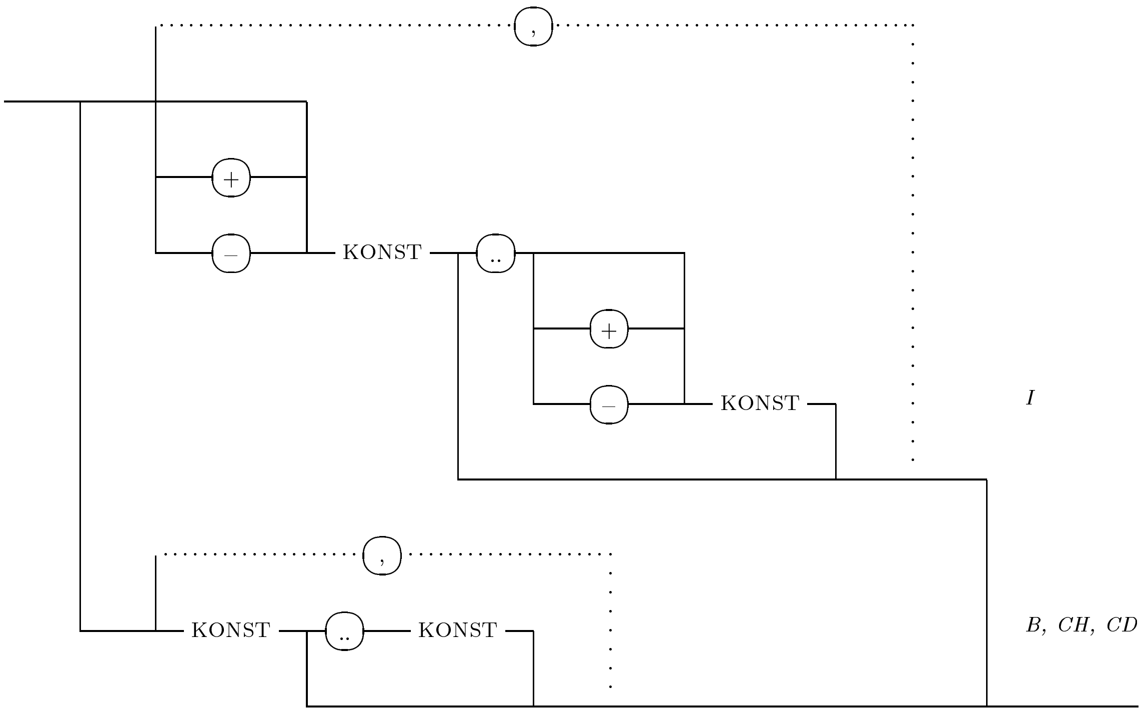
P11 DATENSATZLISTE (DSLISTE)



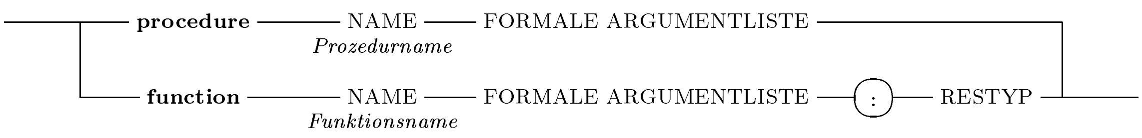
P12 VARIABLENVEREINBARUNG



P13 AUSWAHL

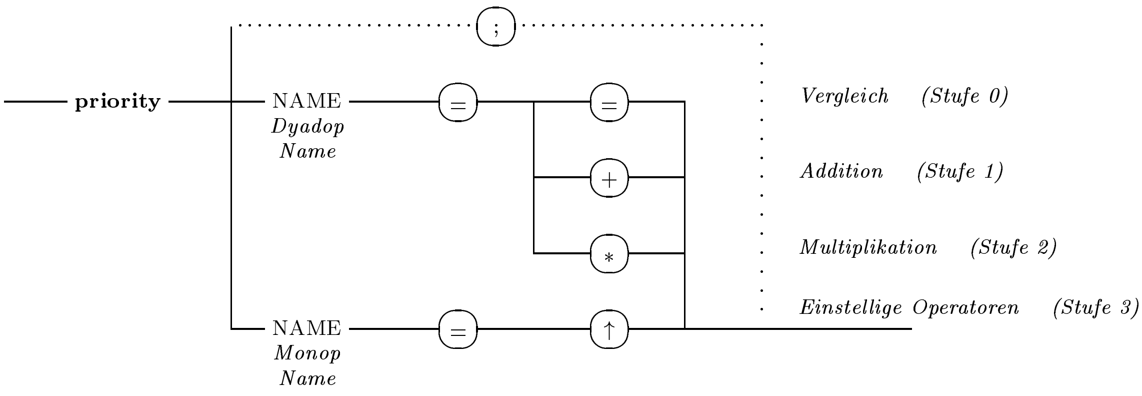


P14 PROZEDUR FUNKTIONSKOPF (PFKOPF)

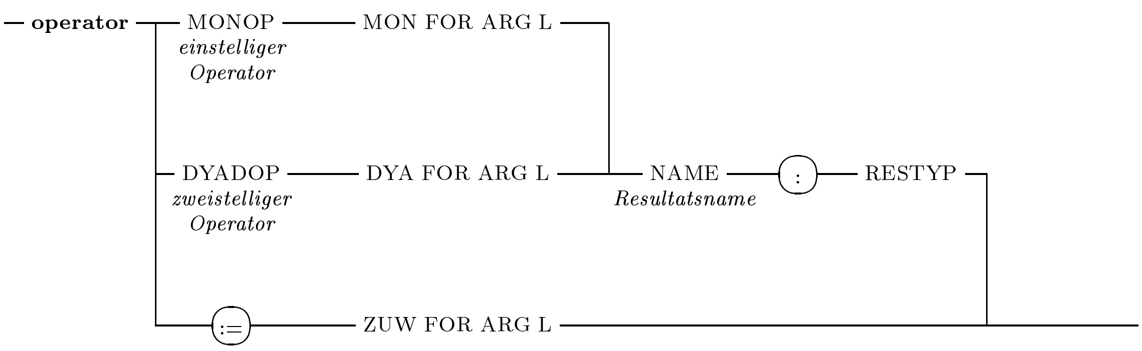


Nach Standard-PASCAL ist bei Vereinbarung einer forward deklarierten Prozedur bzw. Funktion auf die Wiederholung der formalen Argumentliste bzw. der formalen Argumentliste und des Resultatstyps zu verzichten.

P15 PRIORITÄTSDEFINITION

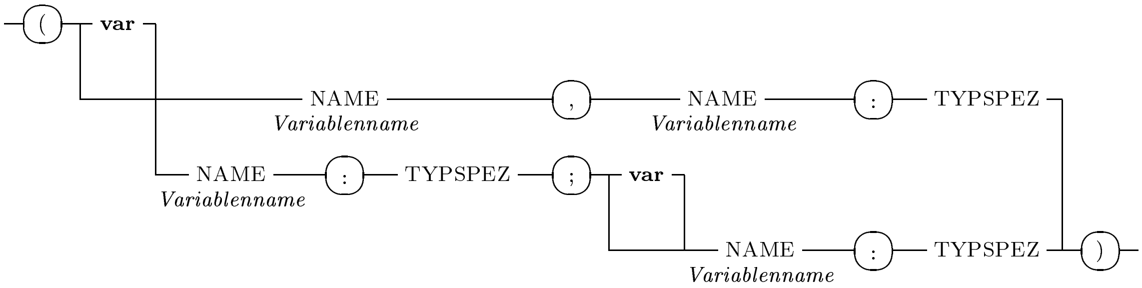


P16 OPERATORKOPF

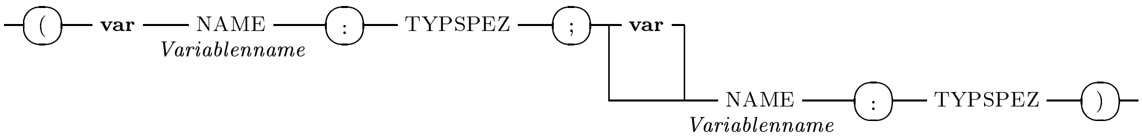


P17 MONADISCHE FORMALE ARGUMENTLISTE (MON FOR ARG L)

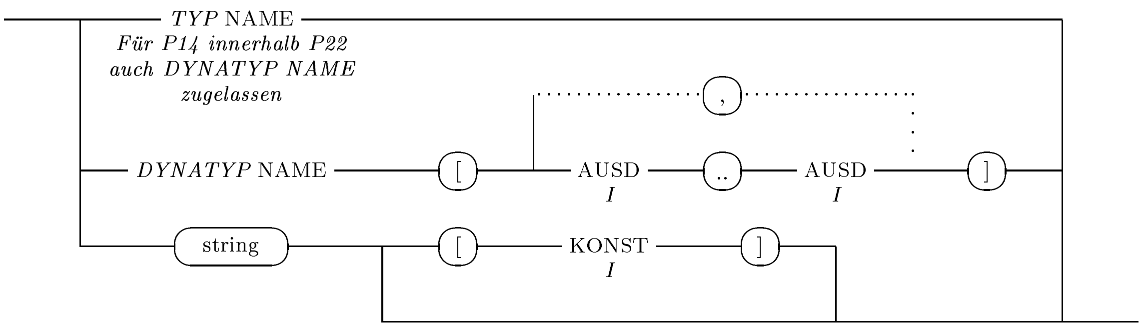
P18 DYADISCHE FORMALE ARGUMENTLISTE (DYA FOR ARG L)



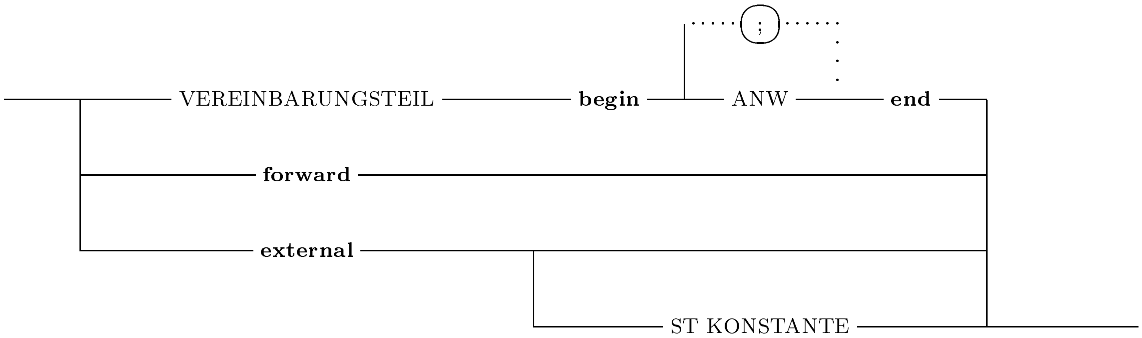
P19 ZUWEISUNGS FORMALE ARGUMENTLISTE (ZUW FOR ARG L)



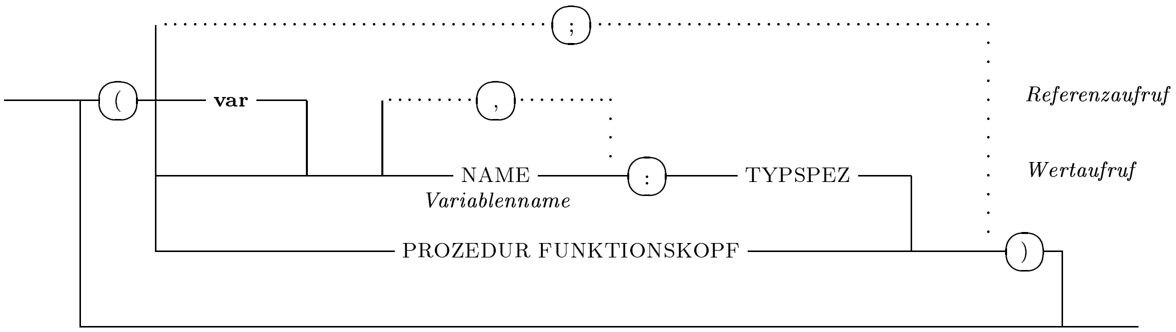
P20 RESULTATSTYP (RESTYP)



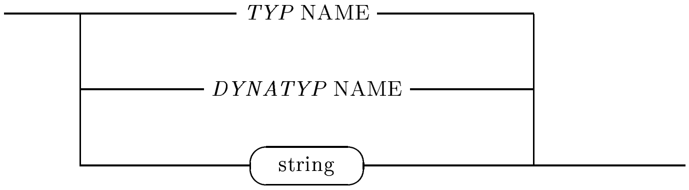
P21 RUMPF



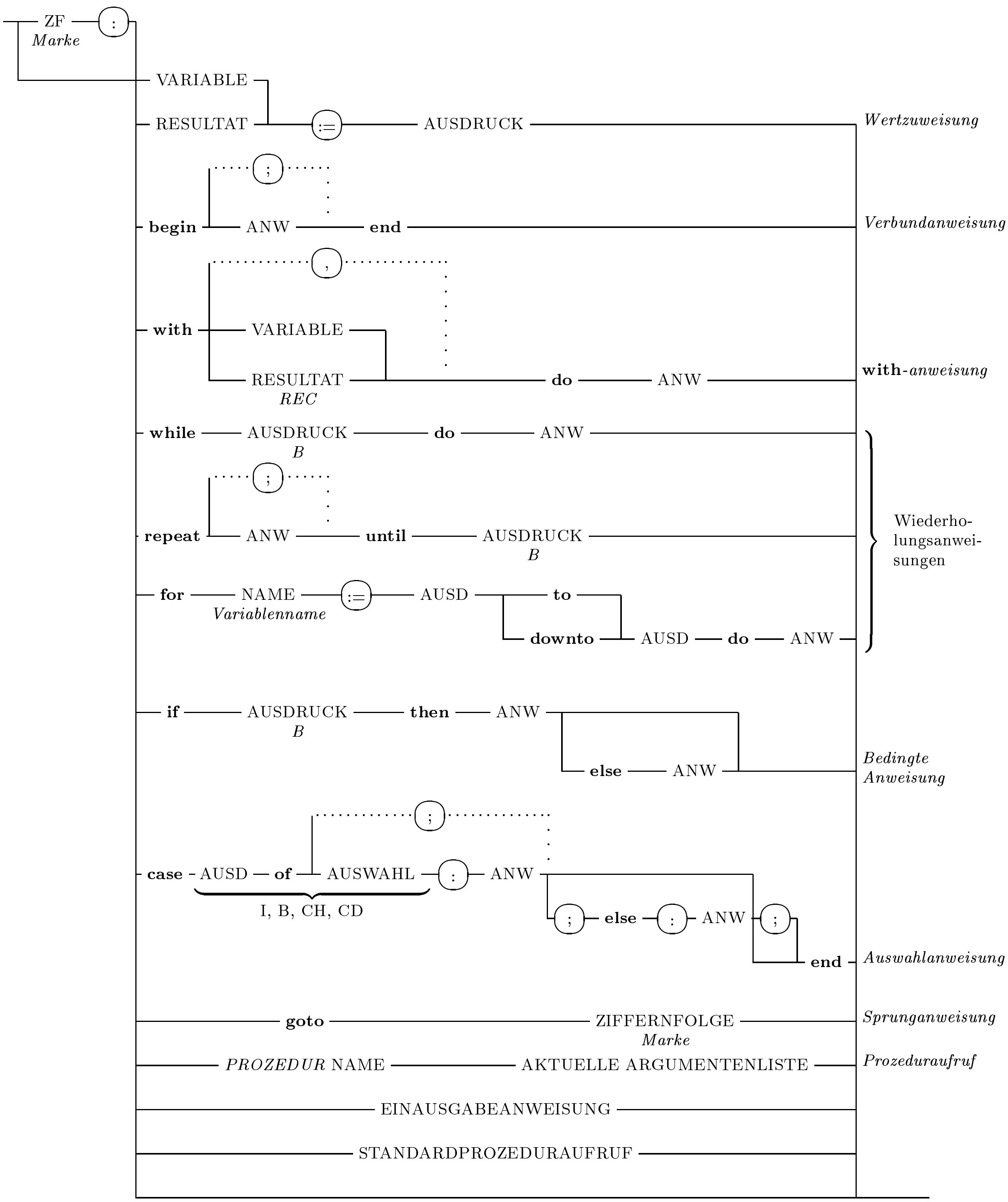
P22 FORMALE ARGUMENTLISTE



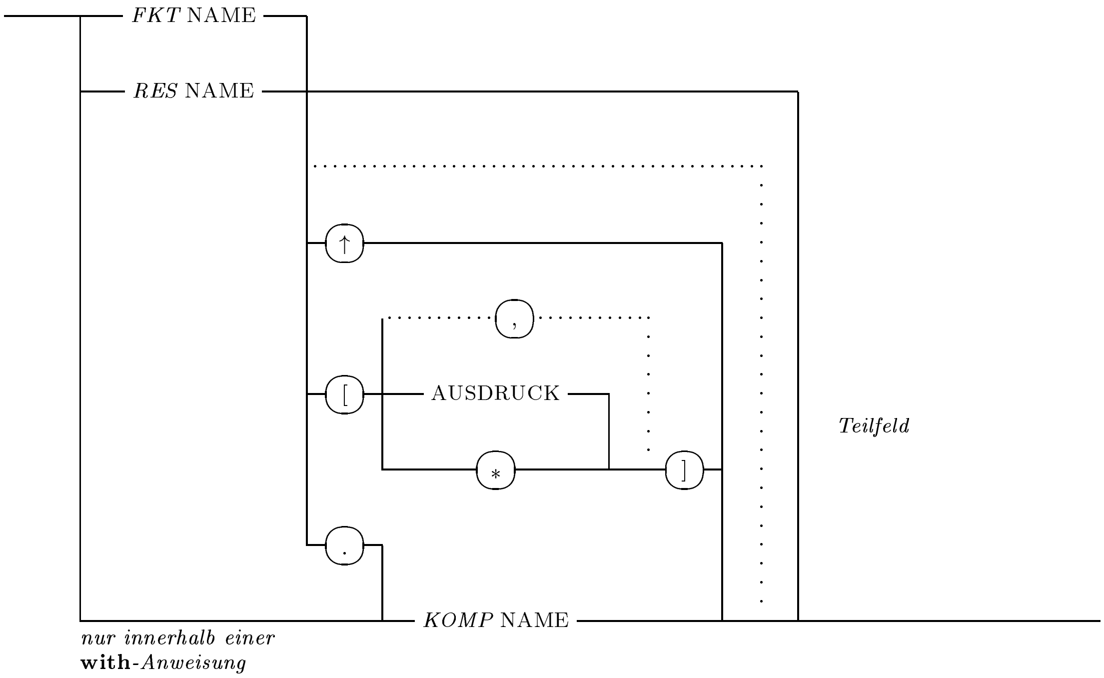
P23 TYPSPEZIFIKATION (TYPSPEZ)



P24 ANWEISUNG (ANW)



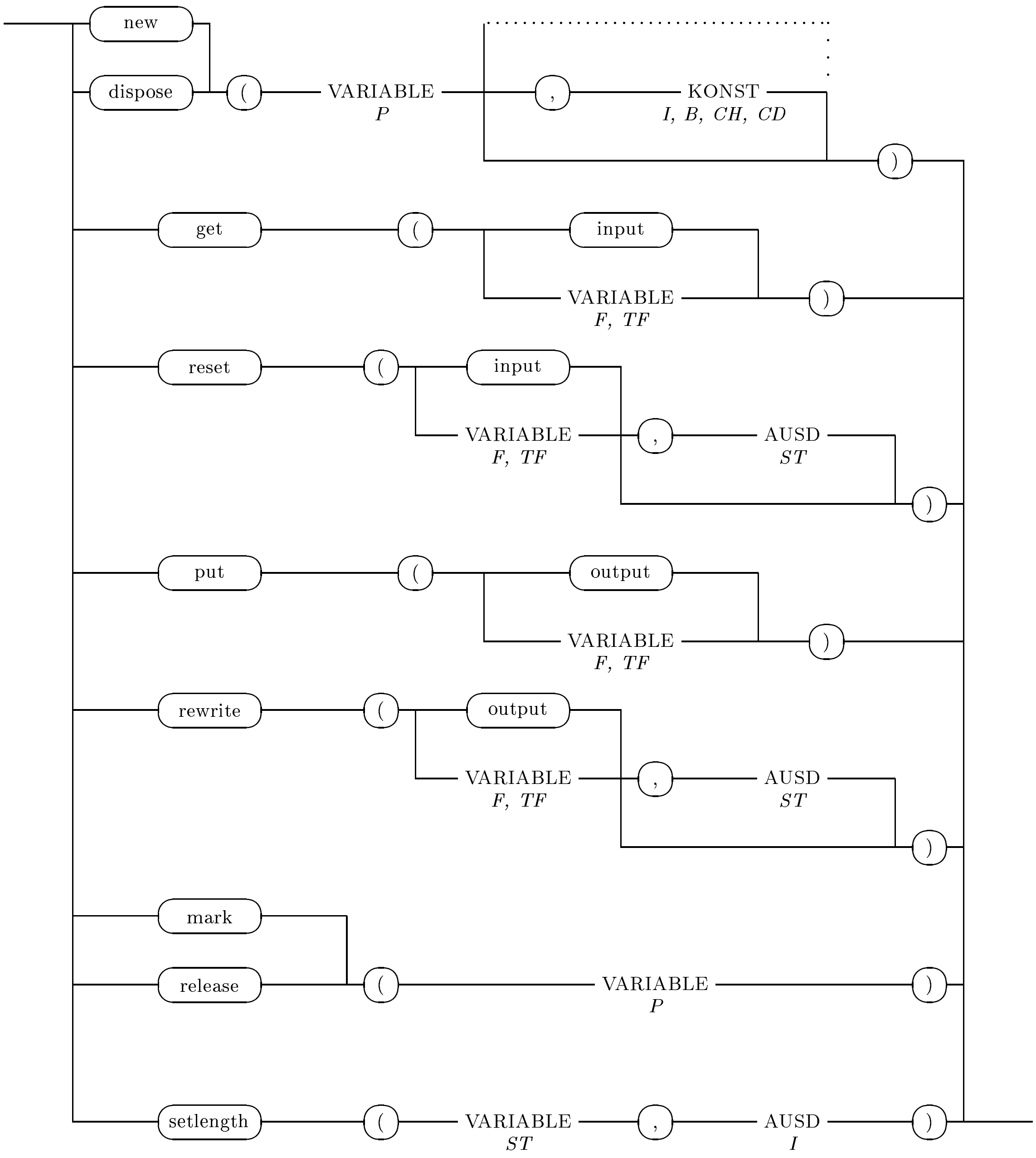
P25 RESULTAT



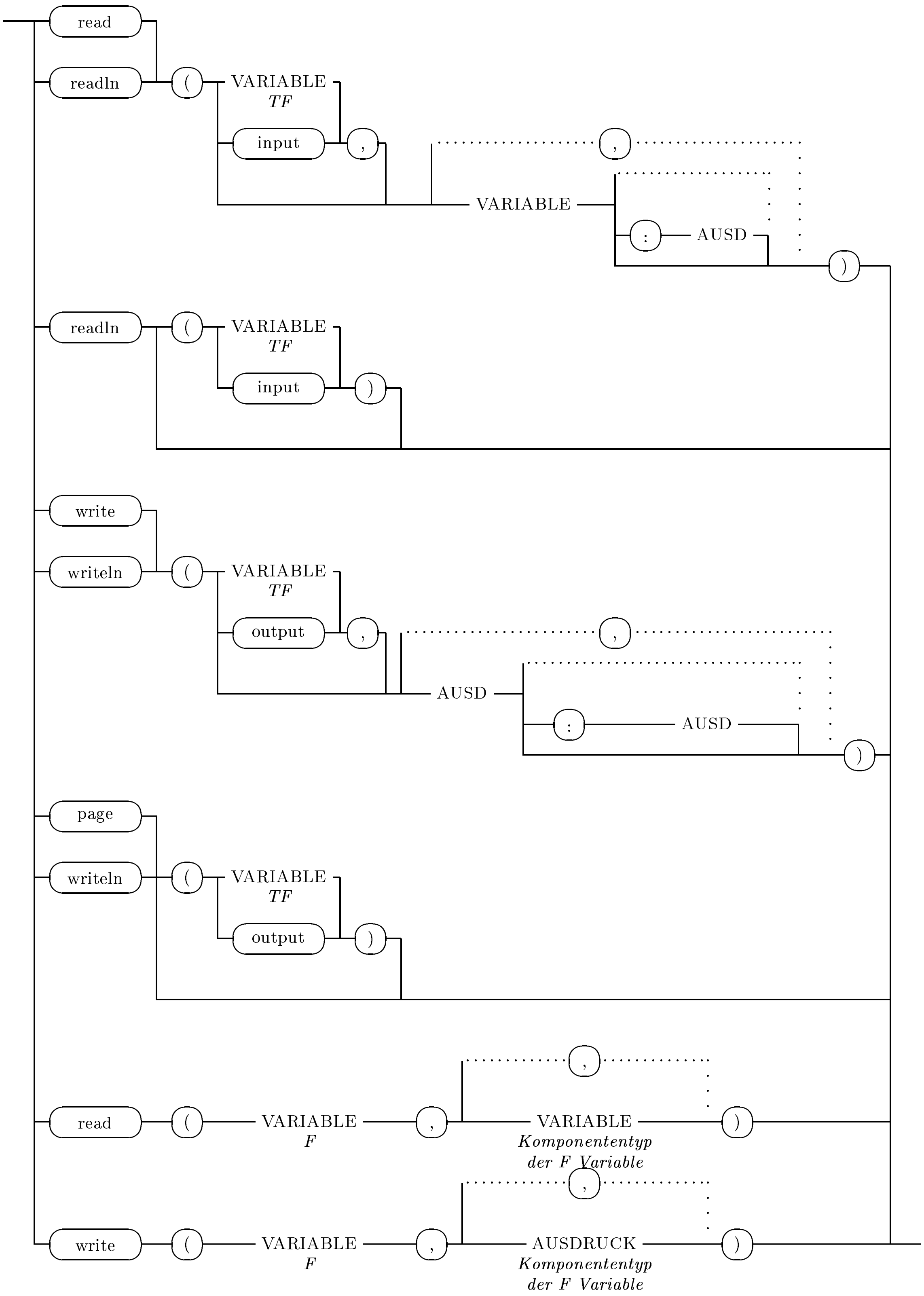
Vordefinierte PASCAL–XSC Komponentennamen

re, im, inf, sup

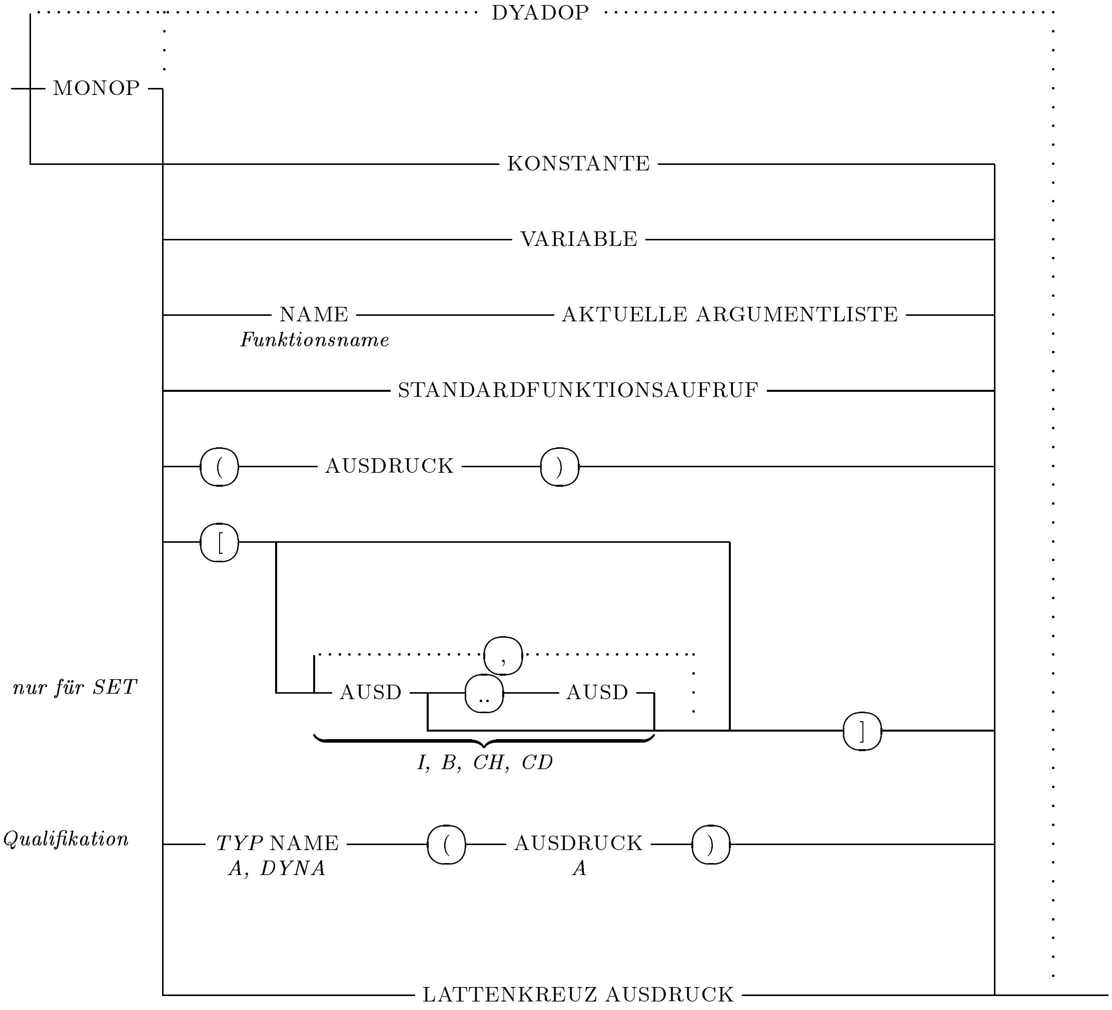
P26 STANDARDPROZEDURAUFRUF



P27 EINAUSGABEANWEISUNG

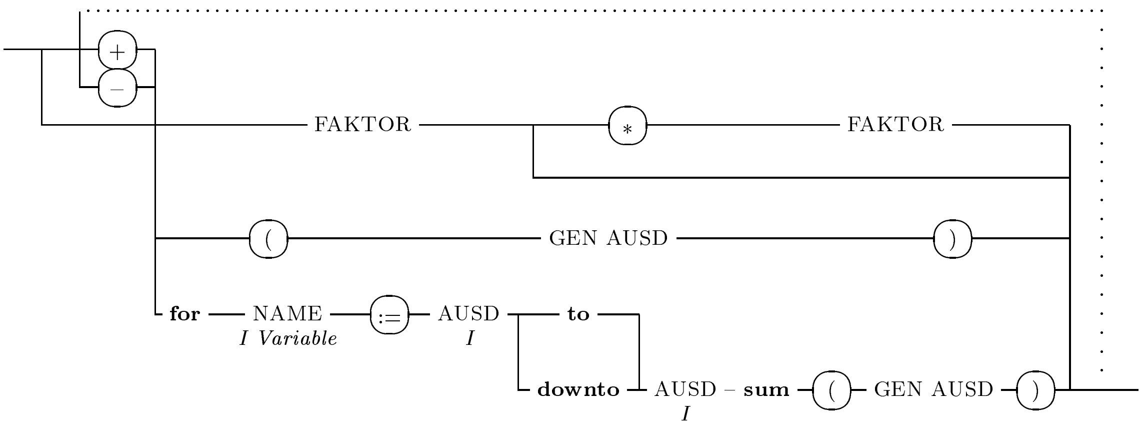


P28 AUSDRUCK (AUSD)



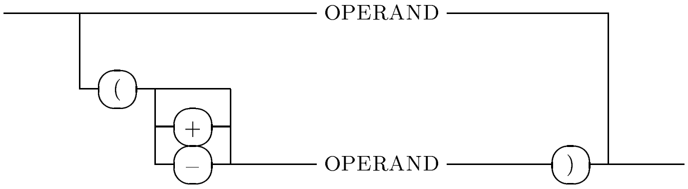
P29 LATTENKREUZ AUSDRUCK

P30 GENAU AUSZUWERTENDER AUSDRUCK (GEN AUSD)

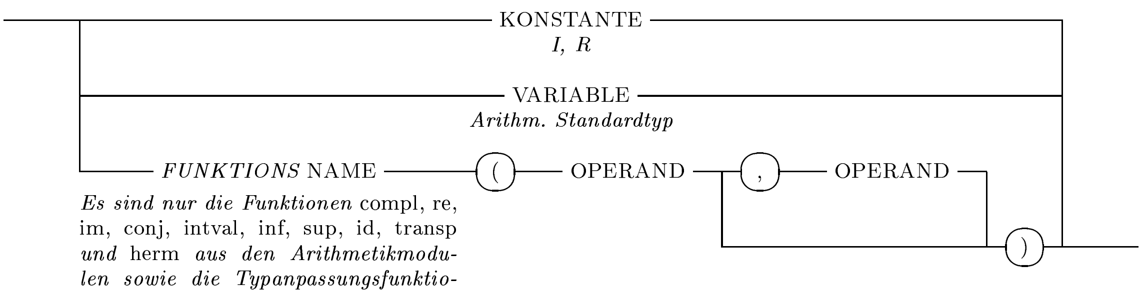


Alle Summanden müssen von gleicher Struktur (Skalar, Vektor oder Matrix) und gleicher Dimension sein. In der Laufanweisung sind keine expliziten Lattenkreuzausdrücke im I AUSD erlaubt.

P31 FAKTOR

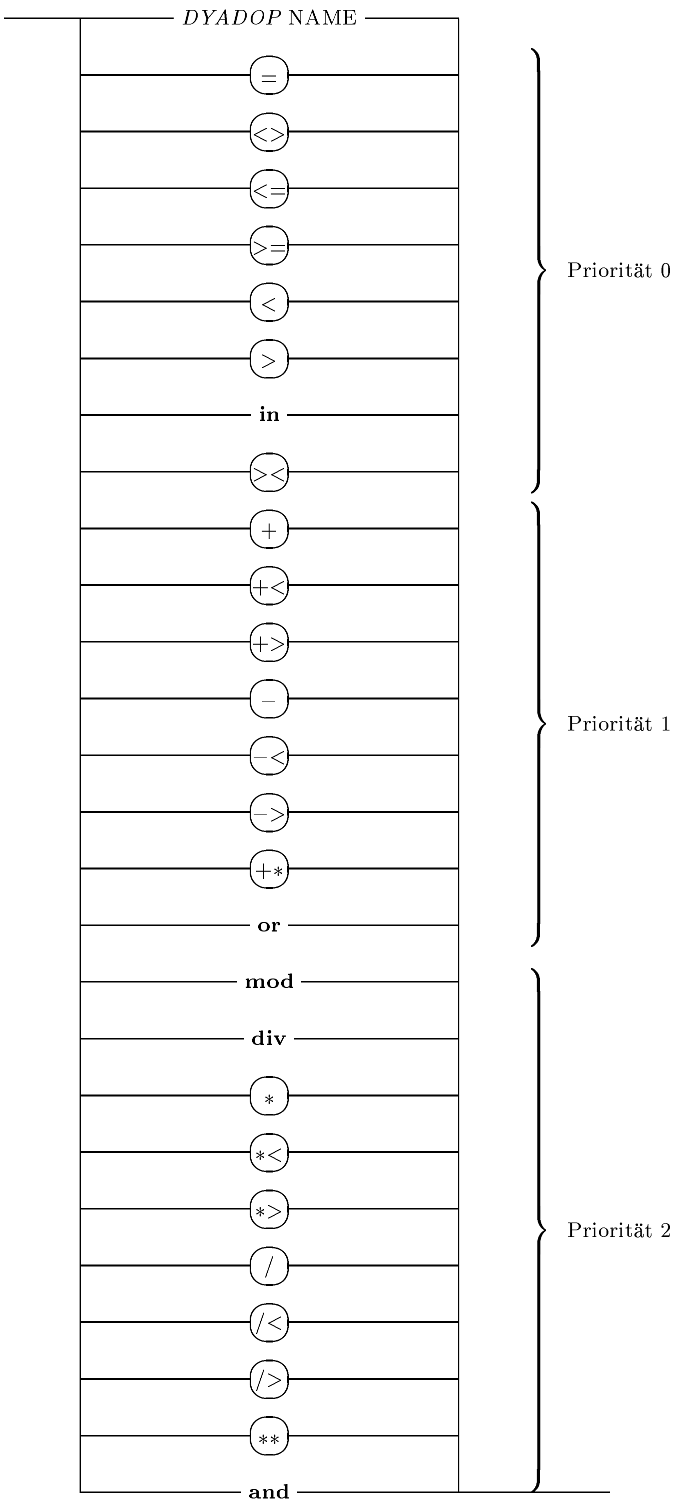


P32 OPERAND

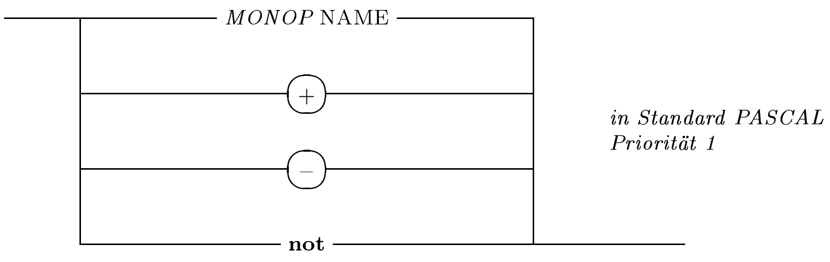


Es sind nur die Funktionen compl, re, im, conj, intval, inf, sup, id, transp *und* herm *aus den Arithmetikmodulen sowie die Typanpassungsfunktionen* rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix *zugelassen (vgl. Abschnitt 2.4.4).*

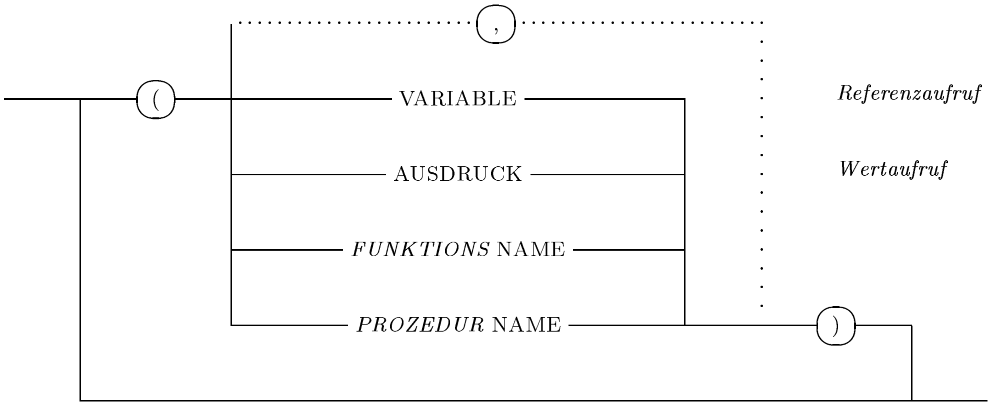
P33 DYADISCHER OPERATOR (DYADOP)



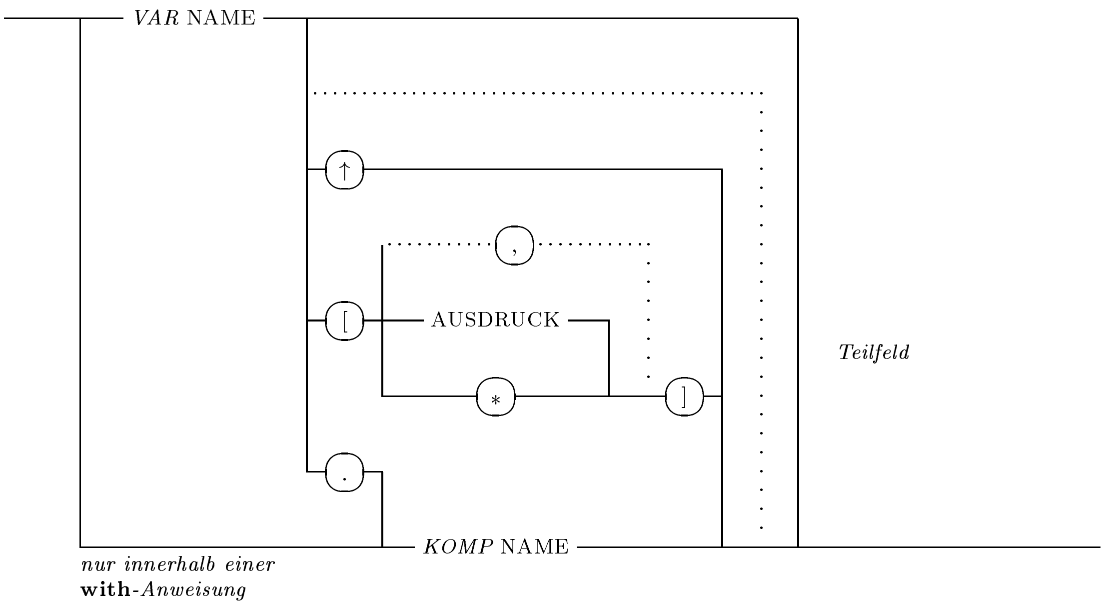
P34 MONADISCHER OPERATOR (MONOP) Priorität 3



P35 AKTUELLE ARGUMENTLISTE



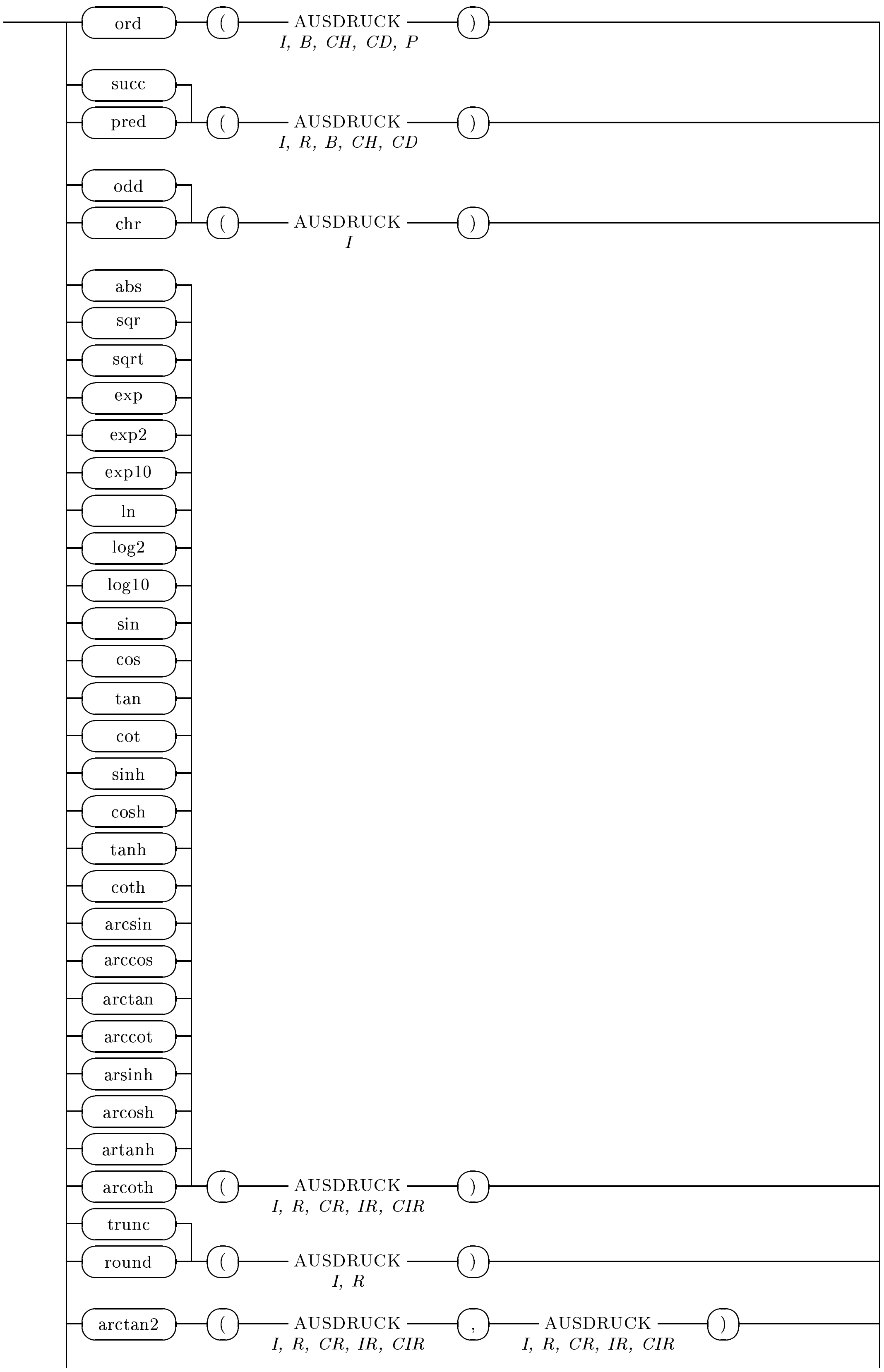
P36 VARIABLE (VAR)



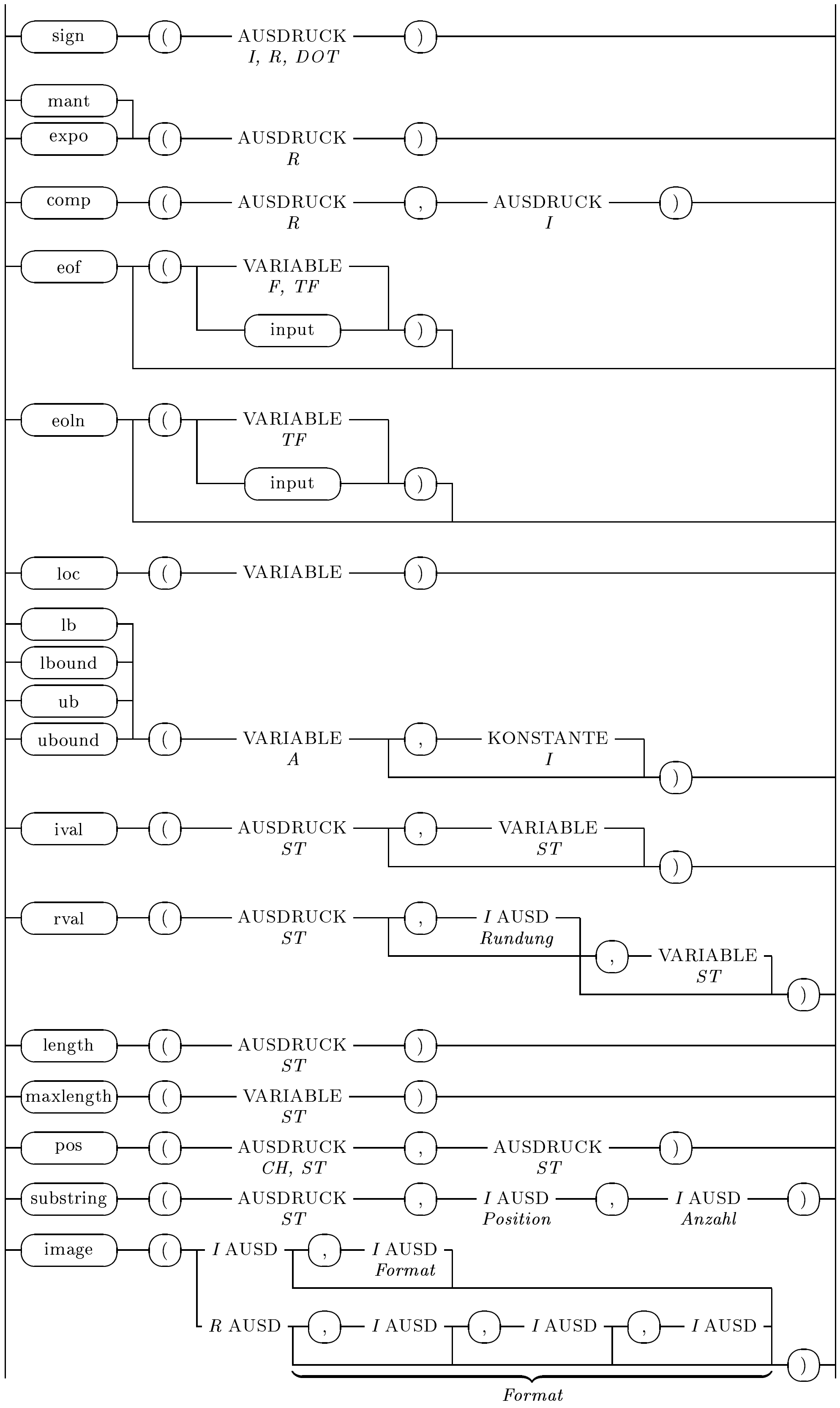
Vordefinierte PASCAL–XSC Komponentennamen

re, im, inf, sup

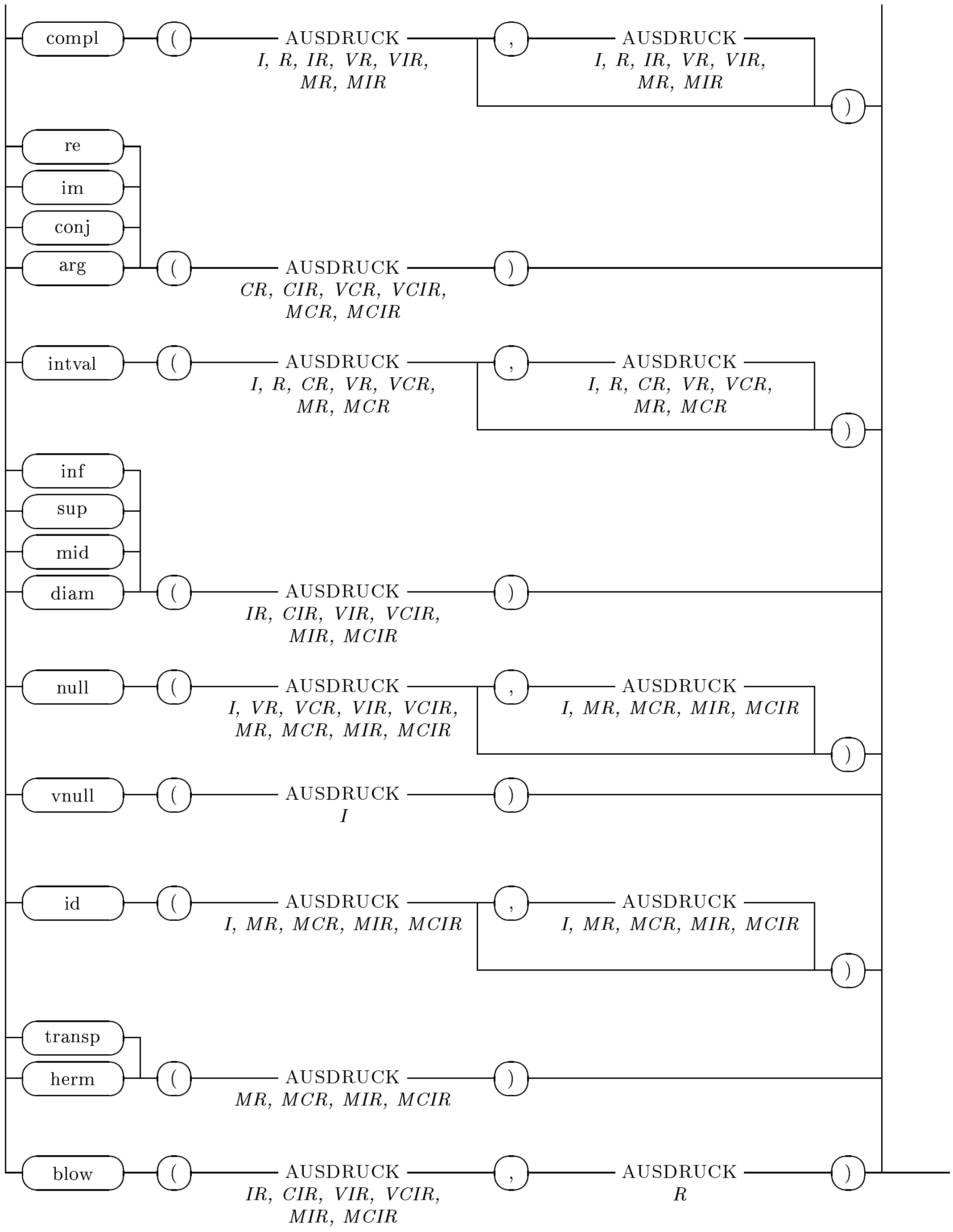
P37 STANDARDFUNKTIONSAUFRUF



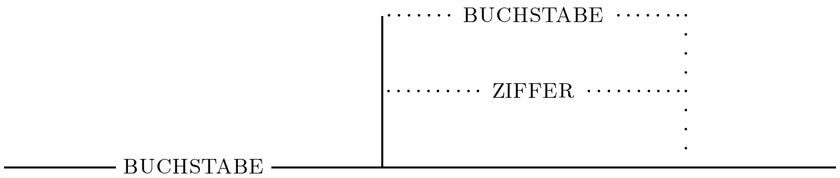
P37 STANDARDFUNKTIONSAUFRUF (Fortsetzung)



P37 STANDARDFUNKTIONSAUFRUF (Fortsetzung, unter Verwendung der Arithmetikmodule)



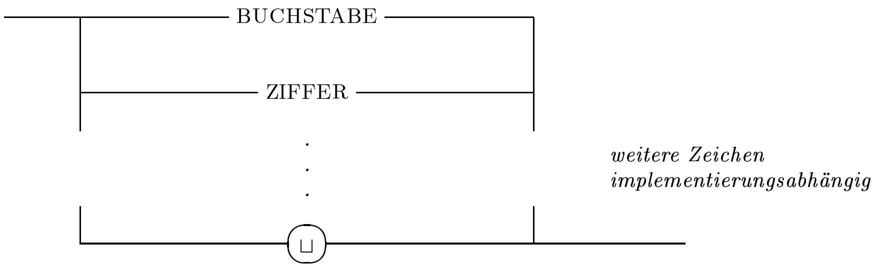
P38 NAME



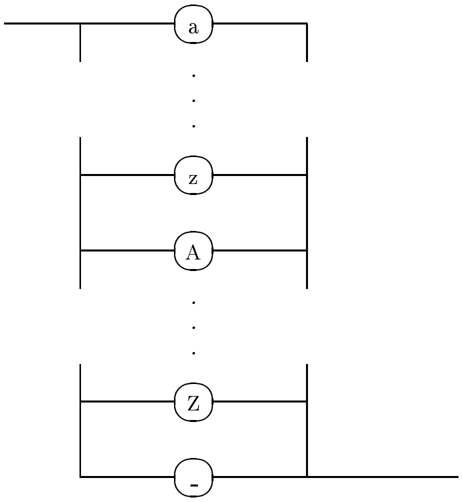
P39 ZIFFERNFOLGE (ZF)

P40 HEXZIFFERNFOLGE (HZF)

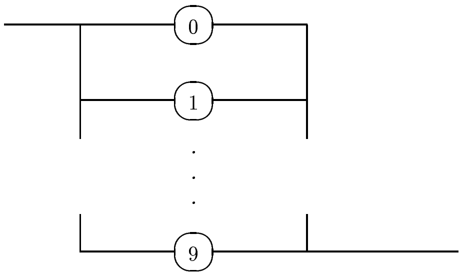
P41 ZEICHEN



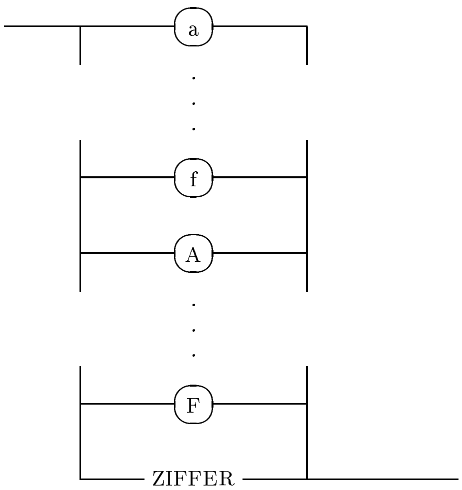
P42 BUCHSTABE



P43 ZIFFER



P44 HEXZIFFER



Anhang B

Verzeichnisse

B.1 Syntaxdiagramme

B.2 Wortsymbole

and **array**

begin

case **const**

div **do** **downto** **dynamic**

else **end** **external**

file **for** **forward** **function**

global **goto**

if **in**

label

mod **module**

nil **not**

of **operator** **or**

packed **priority** **procedure** **program**

record **repeat**

set **sum**

then **to** **type**

until **use**

var

while **with**

B.3 Standardnamen

Im folgenden sind neben den Standardnamen (Bezeichnern) des Sprachkerns auch die in den Arithmetikmodulen definierten aufgeführt. Es handelt sich dabei ausnahmslos um Namen von zusätzlichen Standardfunktionen und Transferfunktionen, die durch *kursive Schrift* gekennzeichnet werden.

Konstanten	false maxint true
Typen	boolean char cimatrix cinterval civector cmatrix complex cvector dotprecision imatrix integer interval ivector real rmatrix rvector string text
Variablen	input output
Komponentennamen	im inf re sup
Funktionen	abs arccos arccot arcosh arcoth arcsin arctan arctan2 *arg* arsinh artanh *blow* chr comp *compl* *conj* cos cosh cot coth *diam* eof eoln exp exp2 exp10 expo *herm* *id* *im* image *inf* *intval* ival lb lbound length ln loc log2 log10 mant maxlength *mid* *null* odd ord pos pred *re* round rval sign sin sinh sqr sqrt substring succ *sup* tan tanh *transp* trunc ub ubound *vnull*

Prozeduren	dispose
	get
	mark
	new
	page put
	read readln release reset rewrite
	write writeln

B.4 Operatoren

Die nachfolgenden Tabellen geben eine Übersicht über die Operatoren des Sprachkerns und der Arithmetikmodule.

B.4.1 Grundlegende Operatoren

linker Operand \ rechter Operand	integer	boolean	char	string	**set**
monadisch	$+, -$	**not**			
integer	$+, -, *, /$, **div**, **mod**, $\vee$				**in**
boolean		**or**, **and**, =, <>, <=, >=			**in**
char			$+$ $\vee$	$+$ $\vee$ **in**	**in**
string			$+$ $\vee$	$+$ $\vee$ **in**	
set					+, −, *, =, <>, <=, >=
Code					**in**

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

B.4.2 Arithmetische Operatoren

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

[1]) Die Operatoren dieser Zeile sind monadisch (es gibt keinen linken Operanden).

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

[3]) $\circ \in \{+, -, *\}$, wobei der $*$-Operator für das Skalar- bzw. Matrixprodukt steht.

[4]) Der $*$-Operator steht für das Skalar- bzw. Matrixprodukt.

$+*$: Intervall-Hülle

$**$: Schnittmenge

B.4.3 Vergleichsoperatoren für die arithmetischen Standardtypen

rechter Operand / linker Operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
integer real complex	=, <>, <=, <, >=, >	**in** =, <>				
interval cinterval	=, <>	**in**, ><,[1] =, <>, <=, <, >=, >				
rvector cvector			=, <>, <=, <, >=, >	**in** =, <>		
ivector civector			=, <>	**in**, ><,[1] =, <>, <=, <, >=, >		
rmatrix cmatrix					=, <>, <=, <, >=, >	**in** =, <>
imatrix cimatrix					=, <>	**in**, ><,[1] =, <>, <=, <, >=, >

[1]) Die Operatoren <= und < stehen für „Teilmenge von“ und „echte Teilmenge von“, >= und > sind die entsprechenden Operatoren für die Obermengenbeziehung

∨ ∈ {=, <>, <, <=, >, >=}

>< : Disjunktheitstest für Intervalle

in : Test ob Punktgröße in einer Intervallgröße liegt oder
Test ob Intervallgröße echt im Innern einer Intervallgröße liegt

B.4.4 Überladungen des Zuweisungsoperators :=

Die folgenden Tabellen geben eine Übersicht über die durch Überladen des Zuweisungsoperators := in den Arithmetikmodulen möglichen Typkombinationen bei der Zuweisung.

Typ der linken Seite	Typ der rechten Seite	Überladungen definiert im
complex	integer real	Modul C_ARI
interval	integer real	Modul I_ARI
cinterval	integer real complex interval	Modul CI_ARI
rvector	integer real	Modul MV_ARI
cvector	integer real complex rvector	Modul MVC_ARI
ivector	integer real interval rvector	Modul MVI_ARI
civector	integer real complex interval cinterval rvector cvector ivector	Modul MVCI_ARI
rmatrix	integer real	Modul MV_ARI
cmatrix	integer real complex rmatrix	Modul MVC_ARI

Typ der linken Seite	Typ der rechten Seite	Überladungen definiert im
imatrix	integer real interval rmatrix	Modul MVI_ARI
cimatrix	integer real complex interval cinterval rmatrix cmatrix imatrix	Modul MVCI_ARI

B.5 Standardfunktionen

Im folgenden wird eine alphabetisch geordnete Übersicht über die Standardfunktionen mit Deklarationsteil (Schnittstelle) und Erläuterungen zur Funktionsweise gegeben. Für die in den Arithmetikmodulen für zusätzliche Datentypen überladenen bzw. neu definierten Funktionen werden die entsprechenden Module angegeben. Für die generischen mathematischen Standardfunktionen wird auf die separate Tabelle am Ende dieses Abschnitts verwiesen. Die Definitions- und Wertebereiche der einzelnen Funktionen können dem Bedienungshandbuch zur jeweiligen Compilerversion entnommen werden.

abs

Siehe Tabelle auf Seite 321

arccos

Siehe Tabelle auf Seite 321

arccot

Siehe Tabelle auf Seite 321

arcosh

Siehe Tabelle auf Seite 321

arcoth

Siehe Tabelle auf Seite 321

arcsin

Siehe Tabelle auf Seite 321

arctan

Siehe Tabelle auf Seite 321

arctan2

Siehe Tabelle auf Seite 321

arg

function arg (c: complex) : real;

Funktion:	Liefert den Argumentwinkel aus der Exponentialdarstellung von c.
def./überl. in:	C_ARI

function arg (c: cinterval) : interval;

Funktion: Liefert das Argumentwinkelintervall aus der Exponentialdarstellung von c.

def./überl. in: CI_ARI

arsinh

Siehe Tabelle auf Seite 321

artanh

Siehe Tabelle auf Seite 321

blow

function blow (x: Typ1; eps: real) : Typ1;

Typ1: *interval, cinterval, ivector, civector, imatrix, cimatrix*

Funktion: Liefert die sogenannte Epsilon-Aufblähung des Intervall-Arguments x (für Array-Typen komponentenweise). *blow* berechnet sich für x vom Typ *interval* aus

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

def./überl. in: I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

chr

function chr (i: integer) : char;

Funktion: Liefert das Zeichen mit der Ordinalzahl i des zugrundeliegenden Zeichensatzes (implementierungsabhängig).

comp

function comp (m: real; e: integer) : real;

Funktion: Komposition einer Mantisse m und eines Exponenten e zu einem Wert $m \cdot b^e$. Die Wertebereiche von b, e und m sind implementierungsabhängig.

conj

function conj (c: Typ1) : Typ1;

Typ1:	*complex, cinterval, cvector, civector, cmatrix, cimatrix*
Funktion:	Konjugation (für Vektor- und Matrix-Typen komponentenweise).
def./überl. in:	C_ARI, CI_ARI, MVC_ARI, MVCI_ARI

cos

Siehe Tabelle auf Seite 321

cosh

Siehe Tabelle auf Seite 321

cot

Siehe Tabelle auf Seite 321

coth

Siehe Tabelle auf Seite 321

diam

function diam (x: Typ1) : ErgTyp;

Typ1:	*interval, cinterval, ivector, civector, imatrix, cimatrix*
ErgTyp:	*real, rvector, rmatrix* je nach Struktur von Typ1.
Funktion:	Liefert den Durchmesser des Arguments (für Array-Typen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

eof

function eof (**var** f: Typ1) : boolean;

Typ1:	*text,* **file of** ...
Funktion:	Liefert *false* falls die aktuelle Komponente der Filevariablen *f* beschrieben ist, andernfalls *true*.

function eof : boolean;

Funktion:	Wie *eof (input)*.

eoln

function eoln (**var** f: text) : boolean;

Funktion:	Liefert *true* falls die aktuelle Komponente der Filevariablen f das Zeilenendezeichen ist, andernfalls *false*.

function eoln : boolean;

Funktion:	Wie *eoln (input)*.

exp

Siehe Tabelle auf Seite 321

exp2

Siehe Tabelle auf Seite 321

exp10

Siehe Tabelle auf Seite 321

expo

function expo (x: real) : integer;

Funktion:	Liefert den Exponenten des Arguments bezogen auf die normalisierte Mantisse und die Basis.

herm

function herm (x: Typ1) : Typ1;

Typ1:	*cmatrix, cimatrix*
Funktion:	Liefert die hermitesche Matrix.
def./überl. in:	MVC_ARI, MVCI_ARI

id

function id (x: Typ1) : rmatrix[lb(x)..ub(x),lb(x,2)..ub(x,2)];

Typ1:	*rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Liefert eine Einheitsmatrix mit den Indexbereichen des Arguments.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

function id (x, y: Typ1) : rmatrix[lb(x)..ub(x),lb(y,2)..ub(y,2)];

Typ1:	*rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Liefert eine Einheitsmatrix mit den Indexbereichen der Produktmatrix $x \cdot y$.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

function id (n: integer) : rmatrix[1..n,1..n];

Funktion:	Liefert eine quadratische $n \times n$ Einheitsmatrix.
def./überl. in:	MV_ARI

function id (n1, n2: integer) : rmatrix[1..n1,1..n2];

Funktion:	Liefert eine rechteckige *n1* $\times$ *n2* Einheitsmatrix.
def./überl. in:	MV_ARI

image

function image (i: integer) : string;

Funktion: Liefert den *integer*-Wert *i* als String mit einer der Standardausgabe für den Typ *integer* entsprechenden Länge (eventuell mit führenden Leerzeichen).

function image (i: integer; width: integer) : string;

Funktion: Liefert den *integer*-Wert *i* als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen).

function image (r: real) : string;

Funktion: Liefert den *real*-Wert *r* als String mit einer der Standardausgabe für den Typ *real* entsprechenden Länge (eventuell mit führenden Leerzeichen).

function image (r: real; width: integer) : string;

Funktion: Liefert den *real*-Wert *r* als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen).

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

Funktion: Liefert den *real*-Wert *r* als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen) und *fracs* Nachkommastellen.

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

Funktion: Liefert den *real*-Wert *r* als String mit mindestens *width* Zeichen (eventuell mit führenden Leerzeichen), *fracs* Nachkommastellen und entsprechend *round* gerundet (<0 nach unten, =0 nächstliegend, >0 nach oben).

ival

function ival (s: string) : integer;

Funktion: Liefert den in *s* stehenden *integer*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird ebenfalls vernachlässigt.

function ival (s: string; **var** rest: string) : integer;

Funktion: Liefert den in *s* stehenden *integer*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird über *rest* zurückgegeben.

lb

function lb (**var** a: Typ1; i: integer) : ErgTyp;

Typ1: beliebiger Array-Typ

ErgTyp: Indextyp von Typ1

Funktion: Abkürzende Schreibweise für *lbound*. Liefert die Untergrenze des *i*-ten Indexbereichs der Array-Variablen *a*.

function lb (**var** a: Typ1) : ErgTyp;

Typ1: beliebiger Array-Typ

ErgTyp: Indextyp von Typ1

Funktion: Abkürzende Schreibweise für *lbound*. Liefert die Untergrenze des ersten Indexbereichs der Array-Variablen *a*.

lbound

function lbound (**var** a: Typ1; i: integer) : ErgTyp;

Typ1: beliebiger Array-Typ

ErgTyp: Indextyp von Typ1

Funktion: Liefert die Untergrenze des *i*-ten Indexbereichs der Array-Variablen *a*.

function lbound (**var** a: Typ1) : ErgTyp;

Typ1: beliebiger Array-Typ

ErgTyp: Indextyp von Typ1

Funktion: Liefert die Untergrenze des ersten Indexbereichs der Array-Variablen *a*.

length

function length (s: string) : integer;

Funktion:	Liefert die aktuelle Länge des *string*-Ausdrucks *s*.

ln

Siehe Tabelle auf Seite 321

loc

function loc (**var** x: Typ1) : integer;

Typ1:	beliebig
Funktion:	Liefert die Hauptspeicheradresse der Argumentvariablen.

log2

Siehe Tabelle auf Seite 321

log10

Siehe Tabelle auf Seite 321

mant

function mant (x: real) : real;

Funktion:	Liefert die normalisierte Mantisse m (Wertebereich implementierungsabhängig) des Arguments.

maxlength

function maxlength (**var** s: string) : integer;

Funktion:	Liefert die maximale Länge der *string*-Variablen *s*.

mid

function mid (x: Typ1) : ErgTyp;

Typ1:	*interval, cinterval, ivector, civector, imatrix, cimatrix*
ErgTyp:	Typ des Infimums (*inf*) bzw. Supremums (*sup*) von Typ1.
Funktion:	Liefert den Mittelpunkt des Arguments (für Array-Typen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

null

function null (x: Typ1) : rvector[lb(x)..ub(x)];

Typ1:	*rvector, cvector, ivector, civector*
Funktion:	Liefert einen Nullvektor mit dem Indexbereich des Arguments.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

function null (x: Typ2) : rmatrix[lb(x)..ub(x),lb(x,2)..ub(x,2)];

Typ2:	*rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Liefert eine Nullmatrix mit den Indexbereichen des Arguments.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

function null (x, y: Typ2) : rmatrix[lb(x)..ub(x),lb(y,2)..ub(y,2)];

Typ2:	*rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Liefert eine Nullmatrix mit den Indexbereichen der Produktmatrix $x \cdot y$.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

function null (n: integer) : rmatrix[1..n,1..n];

Funktion:	Liefert eine quadratische $n \times n$ Nullmatrix.
def./überl. in:	MV_ARI

function null (n1, n2: integer) : rmatrix[1..n1,1..n2];

Funktion:	Liefert eine rechteckige *n1* $\times$ *n2* Nullmatrix.
def./überl. in:	MV_ARI

odd

function odd (i: integer) : boolean;

Funktion:	Liefert *true* falls *i* eine ungerade Zahl ist, andernfalls *false*.

ord

function ord (x: Typ1) : integer;

Typ1:	*integer, boolean, char*, Aufzählungstyp, Pointertyp
Funktion:	Liefert die Ordinalzahl des Arguments, bei Pointertyp den Wert des Zeigers.

pos

function pos (s1, s2: string) : integer;

Funktion: Liefert die Position des ersten Auftretens von *s1* in *s2*. Für *s1* sind auch *char*-Werte (einkomponentige Strings) zugelassen.

pred

function pred (x: Typ1) : Typ1;

Typ1: *integer, real, boolean, char*, Aufzählungstyp

Funktion: Liefert den Vorgänger des Arguments.

round

function round (x: Typ1) : integer;

Typ1: *integer, real*

Funktion: Rundung zur nächstgelegenen *integer*-Zahl; *real*-Werte genau in der Mitte zwischen zwei *integer*-Zahlen werden zur betragsmäßig größeren gerundet. Es ist

$$\text{round (x)} = \text{sign (x)} * \text{trunc (abs (x)} + 0.5)$$

rval

function rval (s: string) : real;

Funktion: Liefert den in *s* stehenden *real*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird ebenfalls vernachlässigt.

function rval (r: string; **var** rest: string) : real;

Funktion: Liefert den in *s* stehenden *real*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird über *rest* zurückgegeben.

function rval (s: string; round: integer) : real;

Funktion: Liefert den in *s* stehenden *real*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird ebenfalls vernachlässigt. Die Rundung erfolgt gemäß *round* (-1 zur nächstkleineren, 0 zur nächstgelegenen, $+1$ zur nächstgrößeren *real*-Zahl), falls der Wert nicht exakt darstellbar ist.

function rval (r: string; round: integer; **var** rest: string) : real;

Funktion:	Liefert den in *s* stehenden *real*-Wert. Dabei werden führende Leerzeichen überlesen, ein eventuell vorhandener Reststring wird über *rest* zurückgegeben. Die Rundung erfolgt gemäß *round* (-1 zur nächstkleineren, 0 zur nächstgelegenen, $+1$ zur nächstgrößeren *real*-Zahl), falls der Wert nicht exakt darstellbar ist.

sign

function sign (x: Typ1) : integer;

Typ1:	*integer, real, dotprecision*
Funktion:	Liefert das Vorzeichen des Argmuents (-1 für negative Werte, 1 für positive Werte, 0 falls Argument $= 0$).

sin

Siehe Tabelle auf Seite 321

sinh

Siehe Tabelle auf Seite 321

sqr

Siehe Tabelle auf Seite 321

sqrt

Siehe Tabelle auf Seite 321

substring

function substring (s: string; pos, anz: integer) : string;

Funktion:	Liefert den *anz* Zeichen langen Teilstring aus *s* ab dem *pos*-ten Zeichen. Wenn *pos* größer als die aktuelle Länge von *s* ist, wird ein leerer String geliefert. Wenn *s* kürzer als *pos* + *anz* ist, werden nur die Zeichen zurückgeliefert, die *s* ab der durch *pos* angegebenen Position (für *pos* < 1 ab der ersten Position) enthält.

succ

function succ (x: Typ1) : Typ1;

Typ1:	*integer, real, boolean, char*, Aufzählungstyp
Funktion:	Liefert den Nachfolger des Arguments.

tan

Siehe Tabelle auf Seite 321

tanh

Siehe Tabelle auf Seite 321

transp

function transp (x: Typ1) : Typ1;

Typ1:	*rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Liefert die transponierte Matrix.
def./überl. in:	MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

trunc

function trunc (x: Typ1) : integer;

Typ1:	*integer, real*
Funktion:	Rundung zu einer *integer*-Zahl durch Abschneiden der Nachkommastellen.

ub

function ub (**var** a: Typ1; i: integer) : ErgTyp;

Typ1:	beliebiger Array-Typ
ErgTyp:	Indextyp von Typ1
Funktion:	Abkürzende Schreibweise für *ubound*. Liefert die Obergrenze des *i*-ten Indexbereichs der Array-Variablen *a*.

function ub (**var** a: Typ1) : ErgTyp;

Typ1:	beliebiger Array-Typ
ErgTyp:	Indextyp von Typ1
Funktion:	Abkürzende Schreibweise für *ubound*. Liefert die Obergrenze des ersten Indexbereichs der Array-Variablen *a*.

ubound

function ubound (**var** a: Typ1; i: integer) : ErgTyp;

Typ1:	beliebiger Array-Typ
ErgTyp:	Indextyp von Typ1
Funktion:	Liefert die Obergrenze des i-ten Indexbereichs der Array-Variablen a.

function ubound (**var** a: Typ1) : ErgTyp;

Typ1:	beliebiger Array-Typ
ErgTyp:	Indextyp von Typ1
Funktion:	Liefert die Obergrenze des ersten Indexbereichs der Array-Variablen a.

vnull

function vnull (n: integer) : rvector[1..n];

Funktion:	Liefert einen n-komponentigen Nullvektor.
def./überl. in:	MV_ARI

Die mathematischen Standardfunktionen im Überblick

	Funktion	generischer Name	Argumenttyp
1	Betrag	abs	$*$
2	Arkuskosinus	arccos	$*$
3	Arkuskotangens	arccot	$*$
4	Areakosinus	arcosh	$*$
5	Areakotangens	arcoth	$*$
6	Arkussinus	arcsin	$*$
7	Arkustangens	arctan	$*$
8	Areasinus	arsinh	$*$
9	Areatangens	artanh	$*$
10	Kosinus	cos	$*$
11	Kotangens	cot	$*$
12	Hyperbolischer Kosinus	cosh	$*$
13	Hyperbolischer Kotangens	coth	$*$
14	Exponentialfunktion	exp	$*$
15	Exponentialfunktion (zur Basis 2)	exp2	$*$
16	Exponentialfunktion (zur Basis 10)	exp10	$*$
17	Natürlicher Logarithmus (Basis e)	ln	$*$
18	Logarithmus zur Basis 2	log2	$*$
19	Logarithmus zur Basis 10	log10	$*$
20	Sinus	sin	$*$
21	Hyperbolischer Sinus	sinh	$*$
22	Quadrat	sqr	$*$
23	Quadratwurzel	sqrt	$*$
24	Tangens	tan	$*$
25	Hyperbolischer Tangens	tanh	$*$

Das Symbol $*$ in der Argumenttyp-Spalte steht für die Typen *integer*, *real*, *complex*, *interval* und *cinterval*, d. h. die Funktionen sind nicht nur für die Typen *integer* und *real* definiert, sondern werden in den Arithmetikmodulen C_ARI, I_ARI und CI_ARI auch für die Typen *complex*, *interval* und *cinterval* zur Verfügung gestellt.

Auf die Angabe der Schnittstelle bzw. der formalen Deklaration wird verzichtet, da alle aufgeführten Funktionen nur ein Argument vom angegebenen Typ zulassen. Der Ergebnistyp ist in der Regel der Typ des Arguments, was für *integer*-Argumente nur für die Funktionen *abs* und *sqr* gilt, während alle anderen Standardfunktionen beim Aufruf mit *integer*-Argumenten *real*-Ergebnisse liefern.

Zusätzlich zu den in der Tabelle angegebenen mathematischen Standardfunktionen steht die Funktion

arctan2 (x1,x2)

mit zwei Argumenten *x1, x2* vom Typ *real* oder *interval* zur Verfügung, die den Wert

arctan (x1/x2)

liefert.

B.6 Transferfunktionen

Im folgenden wird eine alphabetische Übersicht über die Transferfunktionen zur Wandlung zwischen den arithmetischen Standardtypen gegeben. Neben der Angabe ihres Deklarationsteils (Schnittstelle) und Erläuterungen zur Funktionsweise werden auch die jeweiligen Module angegeben, in denen diese Funktionen definiert bzw. überladen werden.

compl

function compl (x1: Typ1; x2: Typ2) : Typ3;

Typ1:	*real, interval, rvector, ivector, rmatrix, imatrix*
Typ2:	*real, interval, rvector, ivector, rmatrix, imatrix* mit der entsprechenden Struktur (Skalar, Vektor, Matrix) von Typ1
Typ3:	Entsprechender komplexer Typ zu Typ1 bzw. Typ2 (*complex, cinterval, cvector, civector, cmatrix, cimatrix*)
Funktion:	Zusammensetzung der als Argumente *x1* und *x2* angegebenen Realteil- und Imaginärteil-Komponenten zum entsprechenden komplexen Datentyp (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	C_ARI, CI_ARI, MVC_ARI, MVCI_ARI

function compl (x: Typ1) : Typ2;

Typ1:	*real, interval, rvector, ivector, rmatrix, imatrix*
Typ2:	Entsprechender komplexer Typ zu Typ1 (*complex, cinterval, cvector, civector, cmatrix, cimatrix*)
Funktion:	Zusammensetzung des als Argument *x* angegebenen Realteils und des Imaginärteils 0 zum entsprechenden komplexen Datentyp (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	C_ARI, CI_ARI, MVC_ARI, MVCI_ARI

im

function im (c: Typ1) : Typ2;

Typ1:	*complex, cinterval, cvector, civector, cmatrix, cimatrix*
Typ2:	Entsprechender reeller Typ zu Typ1 (*real, interval, rvector, ivector, rmatrix, imatrix*)
Funktion:	Liefert den Imaginärteil des Arguments (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	C_ARI, CI_ARI, MVC_ARI, MVCI_ARI

inf

function inf (i: Typ1) : Typ2;

Typ1:	*interval, cinterval, ivector, civector, imatrix, cimatrix*
Typ2:	Entsprechender reeller bzw. komplexer Typ zu Typ1 (*real, complex, rvector, cvector, rmatrix, cmatrix*)
Funktion:	Liefert die Untergrenze des Argumentintervalls (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

intval

function intval (x1: Typ1; x2: Typ2) : Typ3;

Typ1:	*real, complex, rvector, cvector, rmatrix, cmatrix*
Typ2:	*real, complex, rvector, cvector, rmatrix, cmatrix* mit der entsprechenden Struktur (Skalar, Vektor, Matrix) von Typ1.
Typ3:	Entsprechender Intervalltyp zu Typ1 bzw. Typ2 (*interval, cinterval, ivector, civector, imatrix, cimatrix*)
Funktion:	Zusammensetzung der als Argumente *x1* und *x2* angegebenen Unter- und Obergrenzen-Komponenten zum entsprechenden Intervalldatentyp (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

function intval (x: Typ1) : Typ2;

Typ1:	*real, complex, rvector, cvector, rmatrix, cmatrix*
Typ2:	Entsprechender Intervalltyp zu Typ1 (*interval, cinterval, ivector, civector, imatrix, cimatrix*)
Funktion:	Wandlung des Arguments *x* in ein Punktintervall des entsprechenden Intervalldatentyps (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

re

function re (c: Typ1) : Typ2;

Typ1:	*complex, cinterval, cvector, civector, cmatrix, cimatrix*
Typ2:	Entsprechender reeller Typ zu Typ1 (*real, interval, rvector, ivector, rmatrix, imatrix*)
Funktion:	Liefert den Realteil des Arguments (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	C_ARI, CI_ARI, MVC_ARI, MVCI_ARI

sup

function sup (i: Typ1) : Typ2;

Typ1:	*interval, cinterval, ivector, civector, imatrix, cimatrix*
Typ2:	Entsprechender reeller bzw. komplexer Typ zu Typ1 (*real, complex, rvector, cvector, rmatrix, cmatrix*)
Funktion:	Liefert die Obergrenze des Intervall-Arguments (für Vektor- und Matrixtypen komponentenweise).
def./überl. in:	I_ARI, CI_ARI, MVI_ARI, MVCI_ARI

B.7 Standardprozeduren

Im folgenden wird eine alphabetische Übersicht über die Standardprozeduren (einschließlich der Ein-/Ausgabeanweisungen) mit Deklarationsteil (Schnittstelle) und Erläuterungen zur Funktionsweise gegeben. Für die in den Arithmetikmodulen für zusätzliche Datentypen überladenen bzw. neu definierten Prozeduren werden die entsprechenden Module angegeben.

dispose

procedure dispose (**var** p: Typ1);

Typ1:	beliebiger Pointer-Typ
Funktion:	Freigabe des Speicherplatzes eines durch den Pointer *p* referierten Elements.

procedure dispose (**var** p: Typ1; c1,c2,...,cn: Typ2);

Typ1:	beliebiger Pointer-Typ
Typ2:	*integer, boolean, char,* Aufzählungstyp
Funktion:	Freigabe des Speicherplatzes eines durch den Pointer *p* referierten Elements, wobei die Konstanten *c1* bis *cn* das Ansprechen bestimmter Varianten (bei Variantrecords) ermöglichen.

get

procedure get (**var** f: Typ1);

Typ1:	*text*, **file of** ...
Funktion:	Die der aktuellen Komponente der Filevariablen *f* folgende Komponente wird neue aktuelle Komponente. Die Puffervariable $f\uparrow$ erhält den Wert der aktuellen Komponente zugewiesen.

mark

procedure mark (**var** p: Typ1);

Typ1:	beliebiger Pointer-Typ
Funktion:	Markiert den *heap*, um diesen später mit *release* wieder freizugeben.

new

procedure new (**var** p: Typ1);

Typ1:	beliebiger Pointer-Typ
Funktion:	Erzeugung eines neuen referierten Elements zum Pointer *p*.

procedure new (**var** p: Typ1; c1,c2,...,cn: Typ2);

Typ1:	beliebiger Pointer-Typ
Typ2:	*integer, boolean, char,* Aufzählungstyp
Funktion:	Erzeugung eines neuen referierten Elements zum Pointer *p*, wobei die Konstanten *c1* bis *cn* die Erzeugung bestimmter Varianten (bei Variantrecords) ermöglichen.

page

procedure page (**var** f: text);

Funktion:	Beginnt eine neue Seite in der Ausgabedatei *f*.

procedure page;

Funktion:	Entsprechend *page* (*output*).

put

procedure put (**var** f: Typ1);

Typ1:	*text*, **file of** ...
Funktion:	Die aktuelle Komponente von *f* erhält den Wert der Puffervariablen *f*↑ zugewiesen. Die der aktuellen Komponente der Filevariablen *f* folgende Komponente wird neue aktuelle Komponente.

read

procedure read (**var** f: Typ1; **var** x: Typ2);

Typ1:	*text*, **file of** ...
Typ2:	*integer, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Einlesen einer oder mehrerer Variablen vom Typ *Typ2* von der Datei *f* (je nach Typ sind nach jeder Variablen auch durch Doppelpunkt getrennte Formatspezifikationen erlaubt).
def./überl. in:	C_ARI, I_ARI, CI_ARI, MV_ARI, MVC_ARI, MVI_ARI und MVCI_ARI

procedure read (**var** x: Typ2);

Typ2:	*integer, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Entsprechend *read* (*input*, *x*).

readln

procedure readln (**var** f: text);

Funktion: Beenden einer Eingabezeile durch Einlesen des Zeilenendezeichens.

procedure readln;

Funktion: Entsprechend *readln* (*input*).

procedure readln (**var** f: text; **var** x: Typ2);

Typ2: *integer, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*

Funktion: Entsprechend *read* (*f*, *x*) mit anschließendem *readln (f)*.

procedure readln (**var** x: Typ2);

Typ2: *integer, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*

Funktion: Entsprechend *read* (*x*) mit anschließendem *readln*.

release

procedure release (**var** p: Typ1);

Typ1: beliebiger Pointer-Typ

Funktion: Stellt den mit *mark* markierten alten *heap*-Zustand wieder her, alle seit dem Aufruf von *mark* erzeugten referierten Variablen werden wieder freigegeben. *p* darf seit dem Aufruf von *mark* nicht verändert worden sein.

reset

procedure reset (**var** f: Typ1);

Typ1: *text*, **file of** ...

Funktion: Die Datei *f* wird zum Lesen initialisiert.

procedure reset (**var** f: Typ1; s: string);

Typ1: *text*, **file of** ...

Funktion: Die Datei *f* wird zum Lesen initialisiert, der Filevariablen *f* wird dabei die physikalische Datei mit externem Namen *s* zugeordnet.

rewrite

procedure rewrite (**var** f: Typ1);

Typ1:	*text*, **file of** ...
Funktion:	Die Datei f wird zum Schreiben initialisiert.

procedure rewrite (**var** f: Typ1; s: string);

Typ1:	*text*, **file of** ...
Funktion:	Die Datei f wird zum Schreiben initialisiert, der Filevariablen f wird dabei die physikalische Datei mit externem Namen s zugeordnet.

setlength

procedure setlength (**var** s: Typ1; i: Typ2);

Typ1:	*string[m]* bzw. *string*
Typ2:	*0..m* bzw. *0..M*
Funktion:	Die aktuelle Länge der *string*-Variablen s wird auf den Wert i gesetzt. Es muß gelten $0 \leq i \leq m$ bzw. $0 \leq i \leq M$ mit der implementierungsabhängigen Maximallänge M.

write

procedure write (**var** f: Typ1; x: Typ2);

Typ1:	*text*, **file of** ...
Typ2:	*integer, boolean, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Schreiben eines Ausdrucks oder mehrerer Ausdrücke vom Typ *Typ2* auf die Datei f (je nach Typ sind auch nach jedem Ausdruck durch Doppelpunkt getrennte Formatspezifikationen erlaubt).
def./überl. in:	C_ARI, I_ARI, CI_ARI, MV_ARI, MVC_ARI, MVI_ARI, MVCI_ARI

procedure write (x: Typ2);

Typ2:	*integer, boolean, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*
Funktion:	Entsprechend *write* (*output*, x).

writeln

procedure writeln (**var** f: text);

Funktion: Beenden einer Ausgabezeile durch Ausgabe des Zeilenendezeichens.

procedure writeln;

Funktion: Entsprechend *writeln* (*output*).

procedure writeln (**var** f: text; x: Typ2);

Typ2: *integer, boolean, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*

Funktion: Entsprechend *write(f, x)* mit anschließendem *writeln(f)*.

procedure writeln (x: Typ2);

Typ2: *integer, boolean, char, string, real, complex, interval, cinterval, rvector, cvector, ivector, civector, rmatrix, cmatrix, imatrix, cimatrix*

Funktion: Entsprechend *write* (*x*) mit anschließendem *writeln*.

B.8 #-Ausdrücke

B.8.1 Reelle und komplexe #-Ausdrücke

Syntax: #-Symbol (GEN AUSD)

#-Symbol	Ergebnistyp	Erlaubte Summanden im GEN AUSD
#	*dotprecision*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real* oder *dotprecision* • Produkte vom Typ *integer* oder *real* • Skalarprodukte vom Typ *real*
#* #< #>	*real*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real* oder *dotprecision* • Produkte vom Typ *integer* oder *real* • Skalarprodukte vom Typ *real*
	complex	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *complex* oder *dotprecision* • Produkte vom Typ *integer*, *real* oder *complex* • Skalarprodukte von Typ *real* oder *complex*
	rvector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* • Produkte vom Typ *rvector* (z. B. *rmatrix* * *rvector*, *real* * *rvector* etc.)
	cvector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* oder *cvector* • Produkte vom Typ *rvector* oder *cvector* (z. B. *cmatrix* * *rvector*, *real* * *cvector* etc.)
	rmatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* • Produkte vom Typ *rmatrix*
	cmatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* oder *cmatrix* • Produkte vom Typ *rmatrix* oder *cmatrix*

B.8.2 Reelle und komplexe Intervall-#-Ausdrücke

Syntax: ## (GEN AUSD)

#-Symbol	Ergebnistyp	Erlaubte Summanden im GEN AUSD
##	*interval*	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *interval* oder *dotprecision* • Produkte vom Typ *integer*, *real* oder *interval* • Skalarprodukte vom Typ *real* oder *interval*
	cinterval	• Variablen, Konstanten und spezielle Funktionsaufrufe vom Typ *integer*, *real*, *complex*, *interval*, *cinterval* oder *dotprecision* • Produkte vom Typ *integer*, *real*, *complex*, *interval* oder *cinterval* • Skalarprodukte vom Typ *real*, *complex*, *interval* oder *cinterval*
	ivector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector* oder *ivector* • Produkte vom Typ *rvector* oder *ivector*
	civector	• Variablen und spezielle Funktionsaufrufe vom Typ *rvector*, *cvector*, *ivector* oder *civector* • Produkte vom Typ *rvector*, *cvector*, *ivector* oder *civector*
	imatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix* oder *imatrix* • Produkte vom Typ *rmatrix* oder *imatrix*
	cimatrix	• Variablen und spezielle Funktionsaufrufe vom Typ *rmatrix*, *cmatrix*, *imatrix* oder *cimatrix* • Produkte vom Typ *rmatrix*, *cmatrix*, *imatrix* oder *cimatrix*

Literaturverzeichnis

[1] Alefeld, G. und Herzberger, J.: *Einführung in die Intervallrechnung.* Bibliographisches Institut, Mannheim, 1974.

[2] Alefeld, G. and Herzberger, J.: *Introduction to Interval Computations.* Academic Press, New York, 1983.

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

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

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

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

[7] Böhm, H.: *Auswertung arithmetischer Ausdrücke mit maximaler Genauigkeit.* In [28], 1982.

[8] British Standards Institution: *Specification for Computer Programming Language PASCAL.* BS 6192:1982, UDC 681.3.06, PASCAL:519.682, London, 1982.

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

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

[11] Jensen, K. and Wirth, N.: *PASCAL User Manual and Report.* ISO PASCAL Standard, 3rd ed., Springer, Berlin, 1985.

[12] Kaucher, E., Klatte, R. und Ullrich, Ch.: *Programmiersprachen im Griff – Band 2: PASCAL.* Bibliographisches Institut, Mannheim, 1981.

[13] Kaucher, E., Klatte, R. und Ullrich, Ch., Wolff von Gudenberg, J.: *Programmiersprachen im Griff – Band 4: ADA*. Bibliographisches Institut, Mannheim, 1983.

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

[15] Kaucher, E. and Miranker, W. L.: *Self-Validating Numerics for Function Space Problems*. Academic Press, New York, 1984.

[16] Kaucher, E. and Rump, S. M.: *E-Methods for Fixed Point Equation* $f(x) = x$. In Computing **28**, 1982.

[17] Kießling, I., Lowes, M. und Paulik, A.: *Genaue Rechnerarithmetik - Intervallrechnung und Programmieren mit PASCAL-SC*. Teubner, Stuttgart, 1988.

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

[19] Klatte, R. und Ullrich, Ch.: *Programmiersprachen im Griff – Band 9: MODULA-2*. Bibliographisches Institut, Mannheim, 1988.

[20] Kulisch, U.: *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik*. Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim, 1976.

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

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

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

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

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

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

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

[28] Kulisch, U. und Ullrich, Ch. (Hrsg.): *Wissenschaftliches Rechnen und Programmiersprachen.* Berichte des German Chapter of the ACM, Band 10, Teubner, Stuttgart, 1982.

[29] Mayer, G.: *Grundbegriffe der Intervallrechnung.* In [23], 1989.

[30] Neaga, M.: *PASCAL-SC – Eine PASCAL-Erweiterung für wissenschaftliches Rechnen.* In [23], 1989.

[31] Rall, L. B.: *Automatic Differentiation, Techniques and Applications.* Lecture Notes in Computer Science, No. 12, Springer, Berlin, 1981.

[32] Rump, S. M.: *Lösung linearer und nichtlinearer Gleichungssysteme mit maximaler Genauigkeit.* In [28], 1982.

[33] Rump, S. M.: *Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen.* In Jahrbuch Überblicke Mathematik, Bibliographisches Institut, Mannheim, 1983.

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

[35] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Kester, C. H. A.: *Report on the Algorithmic Language ALGOL-68.* Numerische Mathematik 14, 79 - 218, 1969.

Stichwortverzeichnis