Fachinformationszentrum Karlsruhe, 1990

Programmierpraktikum mit PASCAL–SC

D. Ratz
Institut für Angewandte Mathematik
Universität Karlsruhe (TH)

1 Einleitung

Im Rahmen der vom Institut für Angewandte Mathematik angebotenen Vorlesung **„Programmieren I – Einstieg in die Informatik“** wurde im Sommersemester 1989 zusätzlich zu den bisherigen Parallelveranstaltungen **„Einstieg in die Informatik mit FORTRAN“** bzw. **„mit PASCAL“** erstmals als dritte Parallelvorlesung **„Einstieg in die Informatik mit PASCAL–SC“** angeboten.

Während in dem zur PASCAL-Vorlesung angebotenen Praktikum zwar auch bisher schon mit dem vom Institut für Angewandte Mathematik entwickelten PASCAL–SC Compiler [6] gearbeitet, in der Vorlesung jedoch im wesentlichen nur Standard-PASCAL gelehrt wurde, war es die Zielsetzung der zusätzlichen Veranstaltung, vor allem die über den Standard hinausgehenden Erweiterungen der Sprache PASCAL–SC Version 2 zu behandeln.

Nach einer kurzen und straffen Zusammenfassung von Standard-PASCAL wurden die neuen Konzepte von PASCAL–SC eingeführt und anhand von darauf abgestimmten speziellen Aufgaben im Rahmen des Praktikums an ATARI Mega ST Rechnern eingeübt.

Die Vorlesung richtete sich somit an Hörer aller Fachrichtungen ab dem ersten Semester, die aus der Schule oder sonstigen Lehrveranstaltungen Vorkenntnisse in PASCAL mitbrachten. Den größten Teil der Teilnehmer bildeten im Sommersemester 1989 die Fachrichtungen Physik (ca. 50 %) und Maschinenbau (ca. 25 %).

Im folgenden wird nun zunächst eine knappe Übersicht über die wesentlichen Bestandteile des verwendeten PASCAL–SC-2 Compilers gegeben. Nach einer kurzen Beschreibung des Praktikumsablaufs werden schließlich noch einige Beispielaufgaben (mit Lösungen) aus dem Praktikum vorgestellt.

2 PASCAL–SC

Die Programmiersprache PASCAL–SC wurde mit dem Ziel entwickelt, ein mächtiges Werkzeug für die numerische Lösung wissenschaftlicher Probleme zur Verfügung zu stellen, in dem eine mathematisch fundierte Rechnerarithmetik in allen Räumen des numerischen Rechnens vorhanden ist (vgl. [9], [12]). Eine erste Version des Compilers stand 1980 auf Z80 Rechnern zur Verfügung, und etwa seit 1988 ist eine weiterentwickelte Version 2 des Compilers auf 68000er Prozessoren verfügbar (vgl. [3]), deren wichtigste Neuerungen dynamische Felder und ein Modulkonzept sind.

Im wesentlichen umfaßt die neue, erweiterte PASCAL–SC Version 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
- Optimale (hochgenaue) Arithmetik
- Kontrollierte Rundung
- Optimales (exaktes) Skalarprodukt
- Standarddatentyp *dotprecision* (den Gleitkommabereich überdeckendes Festkommaformat)
- Sogenannte Skalarprodukt-Ausdrücke (Dot Product Expressions) für hochgenaue Ausdrucksauswertung
- Hochgenaue Standardfunktionen
- Arithmetikmodule für Langzahlarithmetik, Intervallarithmetik, komplexe Arithmetik, komplexe Intervallarithmetik sowie Vektor- und Matrixarithmetik über diesen Räumen.

- Module mit Lösungsroutinen für häufig auftretende numerische Probleme wie z. B.
  - Lineare Gleichungssysteme
  - Nichtlineare Gleichungssysteme
  - Eigenwerte und Eigenvektoren
  - Auswertung von arithmetischen Ausdrücken
  - Auswertung von Polynomen und Nullstellenbestimmung
  - Anfangs- und Randwertprobleme von gewöhnlichen Differentialgleichungen

  Alle Routinen liefern automatisch verifizierte Ergebnisse.

Die wichtigsten dieser Konzepte, auf deren Darstellung auch in der Vorlesung besonderer Wert gelegt wurde, sollen nun kurz erläutert werden.

2.1 Universelles Operatorkonzept, allgemeiner Ergebnistyp

PASCAL–SC bietet im Hinblick auf einfache Programmierung eine große Erleichterung durch die Möglichkeit, Funktionen und auch Operatoren mit allgemeinem Ergebnistyp zu vereinbaren (vgl. auch [12]). 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;
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–SC hingegen wäre es aufgrund eines allgemeinen Ergebnistyps möglich eine Funktion `add`

```
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 durch `z:= add(a,add(b,add(c,d)))` berechnen läßt. Implementiert man dagegen in PASCAL–SC den Operator

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

dann 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–SC auch Operatoren mit Namen definieren, muß diesen dann aber in einer vorangestellten Prioritätsvereinbarung eine Priorität zuweisen.

2.2 Überladen von Prozeduren, Funktionen und Operatoren

PASCAL–SC 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 unterschieden durch die Anzahl, den Typ und die Gewichtung der Parameter. Der Ergebnistyp wird *nicht* zur Unterscheidung herangezogen.

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

2.3 Modulkonzept

Das in der Version 2 des PASCAL–SC Compilers integrierte Modulkonzept (vgl. [4]) 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 kann. 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 bzw. 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_poly (...
  ...
  global procedure write_poly (...
  ...
  global operator + (...
  ...
  global operator * (...
  ...
begin
  { Initialisierungsteil des Moduls}
  ...
end. {module poly}
```

2.4 Dynamische Felder, Teilfelder

Das Konzept der dynamischen Felder ermöglicht es, Algorithmen unabhängig von der Dimension des Problems zu implementieren. Die Größe und der Indexbereich von dynamischen Feldern werden erst zur Laufzeit und nicht bereits zur Übersetzungszeit festgelegt. In PASCAL–SC 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 qualifiziert 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(...)* zur Verfügung. Eine Polynommultiplikation kann dynamisch z. B. mittels

```
operator * (a,b:polynom) rm: polynom[lbound(a)..2*ubound(a)];
  var i,j: integer;
      r  : polynom[lbound(a)..2*ubound(a)];
  begin
    for i:= lbound(a) to 2*ubound(a) do
        r[i]:= 0;
    for i:= lbound(a) to ubound(a) do
      for j:= lbound(a) to ubound (a) do
        r[i+j]:= r[i+j] + a[i] * b[j];
    rm:= r;
  end;
```

realisiert werden.

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

```
program dynamik (input,output);
...
type polynom = dynamic array [*] of real;
...
operator * (a,b:polynom)...
...
```

```
procedure write_poly (p: polynom);
...
procedure main (n : integer);
  var
    p,q,s : polynom[0..n];
    r     : polynom[0..2*n];
  begin
    ...
    r:= p * q;
    write('p*q = '); write_poly(r);writeln;
    ...
  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 }
```

2.5 Stringkonzept

In die Sprache PASCAL–SC wurde ein String-Konzept integriert, das die Handhabung von Zeichenketten variabler Längen ermöglicht. Es ermöglicht eine einfache Vereinbarung, die Ein-/Ausgabe von Strings bzw. String-Ausdrücken und die Verwendung von speziellen Standardfunktionen und Operatoren zur String-Manipulation.

2.6 Arithmetik und Rundung

Die Menge der Standardoperatoren für *real*-Zahlen ist in PASCAL–SC gegenüber Standard-PASCAL erweitert um die Rundungsoperatoren $\circ<$ bzw. $\circ>$, $\circ \in \{+, -, *, /\}$ für die Verknüpfungen mit anschließender gerichteter Rundung nach unten bzw. oben.

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.

2.7 Skalarproduktausdrücke

Für die Implementierung von Einschließungsalgorithmen benötigt man die exakte Auswertung von Skalarprodukten. Um dieses zu ermöglichen, wurde in PASCAL–SC der neue Datentyp *dotprecision* eingeführt, der ein den Gleitkommabereich überdeckendes Festkommaformat darstellt, auf dem skalare Ergebnisse ohne Fehler gespeichert werden können.

Weiterhin können Skalarproduktausdrücke in Vektor- und Matrixform mit nur einer einzigen Rundung pro Komponente berechnet werden.

2.8 Abschließende Bemerkungen

Mit den in PASCAL–SC zur Verfügung stehenden Sprachkonzepten können numerische Algorithmen formuliert werden, die hochgenaue und automatisch verifizierte Ergebnisse liefern (vgl. [8]). Aufgrund seiner Eigenschaften, der Maus-gesteuerten Oberfläche und einem Syntax- und Semantik-prüfenden Editor stellt das neue PASCAL–SC-System auch ein exzellentes Ausbildungssystem dar. Ein sehr wesentlicher Vorteil der Sprache PASCAL–SC gerade im Hinblick auf das Praktikum liegt darin, daß die Programme aufgrund des Operatorkonzepts erheblich leichter zu schreiben und auch zu lesen sind, da alle Operationen - auch die in höheren mathematischen Räumen - in mathematischer Notation im Programm wiedergegeben werden können, was die Beispielaufgaben aus dem Praktikum in Abschnitt 4 deutlich machen.

Eine ausführliche Beschreibung der neuen Konzepte findet sich in [1] bzw. [2]. Die neue Version des Compilers wird Ende 1989 fertiggestellt sein, Version 1 für IBM-PC (vgl. [6]) und ATARI ST (vgl. [7]) ist im Buchhandel erhältlich.

3 Ablauf des Praktikums

Zu Beginn des Kurses erhielt jeder Teilnehmer eine Aufgabensammlung mit den im Semester zu bearbeitenden Übungsaufgaben. Aus dieser Sammlung wurden wöchentlich, abhängig vom behandelten Stoff, ein bis zwei Aufgaben zur verpflichtenden Bearbeitung ausgewählt.

Das Rechnerpraktikum selbst fand an ATARI Mega ST Rechnern statt, an denen für jeden Teilnehmer wöchentlich zwei Stunden Rechenzeit zur

Verfügung stand. Die Übungen wurden von Tutoren betreut, die, falls notwendig, Hilfestellung leisteten und die Pflichtaufgaben am Ende des Bearbeitungszeitraums auf ihre Richtigkeit hin überprüften und testierten. Der rechtzeitige Erhalt aller Testate war Voraussetzung für eine Abschlußklausur (ohne Hilfsmittel), die über die Vergabe eines Übungsscheines entschied.

Die im Praktikum verwendeten Aufgaben lassen sich in fünf Kategorien einteilen:

- einführende Aufgaben zum Erlernen der Systembedienung
- Aufgaben zur Vertiefung der neuen Konzepte von PASCAL–SC (Operatorkonzept, Funktionen mit allgemeinem Ergebnistyp, dynamische Felder, Modulkonzept usw.)
- einfache Aufgaben zur Behandlung von Genauigkeitsfragen bei arithmetischen Operationen bzw. numerischen Berechnungen (Verwendung des Datentyps *dotprecision*)
- Aufgaben zu verschiedenen Arithmetiken (Intervallarithmetik, komplexe Arithmetik, Vektor/Matrix-Arithmetik etc.)
- Aufgaben zu physikalischen bzw. ingenieurwissenschaftlichen Anwendungen von Programmiersprachen und numerischen Verfahren

Einige beispielhaft aus dem Praktikum ausgewählte Aufgaben sollen im folgenden Abschnitt einerseits einen Einblick in den Praktikumsinhalt geben und andererseits anhand der Musterlösungen die effiziente Programmierung mit PASCAL–SC demonstrieren. Die mitabgedruckten Ablaufprotokolle machen die Notwendigkeit der angesprochenen Spracherweiterungen für hochgenaues Rechnen deutlich.

4 Einige Aufgaben aus dem Praktikum

Aufgabe 1: 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–SC Programm, das eine 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 mit der Anweisung `use CRARI;` unmittelbar vor dem Vereinbarungsteil ihres Programms das Modul `CRARI` einbinden und nach jedem Einlesen zunächst Ihren Operator `CDIV` und dann den in `CRARI` vordefinierten Operator / aufrufen und die erhaltenen Werte zum Vergleich ausgeben.

c) Testen Sie dies auch mit den den Werten $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ mit

$$x_1 = 1254027132096, \quad y_1 = 886731088897$$
$$x_2 = 886731088897, \quad y_2 = 627013566048 \quad .$$

Im Imaginärteil werden Sie einen deutlichen Unterschied feststellen.

d) Entwerfen Sie einen weiteren Operator `NCDIV`, der unter Verwendung von *dotprecision*-Ausdrücken bessere Ergebnisse liefert als `CDIV`, und vergleichen Sie anschließend in einem Testlauf die drei Operatoren nochmals.

Lösung:

```
program KomplDiv (input,output);

use crari;

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.im:= (z2.re*z1.im - z1.re*z2.im)/nenner;
      res.re:= (z1.re*z2.re + z1.im*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.im:= #*(z2.re*z1.im - z1.re*z2.im)/nenner;
      nres.re:= #*(z1.re*z2.re + z1.im*z2.im)/nenner;
   end;
```

```
begin
  write ('z1 = '); cread (input,z1);
  write ('z2 = '); cread (input,z2);
  write ('z1 cdiv  z2 = '); cwrite (output, z1 cdiv z2); writeln;
  write ('z1 ncdiv z2 = '); cwrite (output, z1 ncdiv z2); writeln;
  write ('z1   /   z2 = '); cwrite (output, z1/z2); writeln;
end.
```

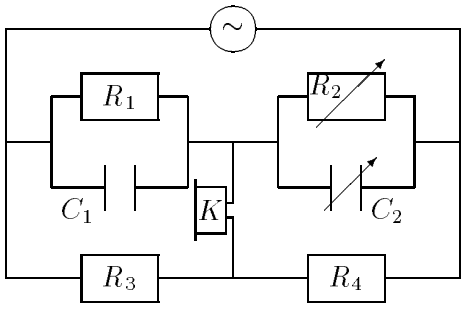
Ablaufprotokoll:

```
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 )
```

Bemerkung: Diese Aufgabe zeigt, daß schon bei so einfachen Problemen wie der komplexen Division die Verwendung einer hochgenauen Ausdrucksauswertung notwendig wird.

Aufgabe 2: Wechselstrom-Meßbrücke



Die unbekannte Kapazität C_1 und der unbekannte Widerstand R_1 können mit Hilfe der o. a. 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:

$$\begin{array}{rcrcl} 9.9\Omega & \leq & R_3 & \leq & 10.1\Omega \\ 6.8\Omega & \leq & R_4 & \leq & 6.9\Omega \end{array} ,$$

für C_2 und R_2 gelten, bedingt durch Meßungenauigkeiten, folgende Abschätzungen:

$$\begin{array}{rcrcl} 40.2\text{F} & \leq & C_2 & \leq & 41.5\text{F} \\ 18.3\Omega & \leq & R_2 & \leq & 19.8\Omega \end{array} .$$

Schreiben Sie ein PASCAL–SC 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.

Lösung:

```
program messbruecke (input,output);
use irari;
var
  c1, c2, r1, r2, r3, r4 : interval;
  d : real;
begin
  write('untere Schranke fuer R3: '); read(r3.inf);
  write('obere  Schranke fuer R3: '); read(r3.sup);
  write('untere Schranke fuer R4: '); read(r4.inf);
  write('obere  Schranke fuer R4: '); read(r4.sup);
  write('untere Schranke fuer C2: '); read(c2.inf);
  write('obere  Schranke fuer C2: '); read(c2.sup);
  write('untere Schranke fuer R2: '); read(r2.inf);
  write('obere  Schranke fuer R2: '); read(r2.sup);
  c1:= r4 * c2 / r3;
  r1:= r3 * r2 / r4;
  writeln;
  writeln('C1 = [',c1.inf,',',c1.sup,']');
  writeln('R1 = [',r1.inf,',',r1.sup,']');
  d := diam(c2) / 2 * 0.1;
  c2:= intval ( c2.inf -< d , c2.sup +> d );
  d := diam(r2) / 2 * 0.1;
  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.inf,',',c1.sup,']');
  writeln('R1 = [',r1.inf,',',r1.sup,']');
end.
```

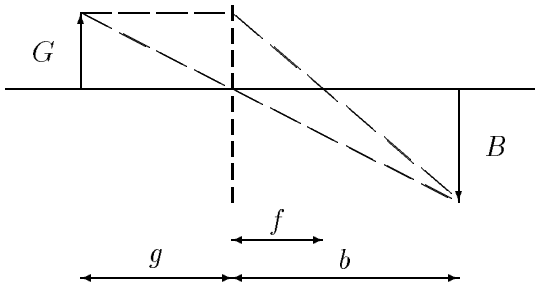
Ablaufprotokoll:

```
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 3: 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 $f_0 = 20\text{cm}$, $b_0 = 25\text{cm}$ und $\triangle f = \triangle b = 1\text{cm}$. Schreiben Sie ein PASCAL–SC 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 irari;

var
  g0, dg, f0, df, b0, db : real;
  g, f, b                : interval;

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);
  write ('f = '); iwrite(output,f); writeln;
  write ('b = '); iwrite(output,b); writeln;
  writeln;
  g0 := 1 / (1/f0 - 1/b0);
  dg := df / sqr(1 - f0/b0) + db / sqr(b0/f0 - 1);
  g  := intval (g0 - dg , g0 + dg);
  write ('g = g0 +/- dg       = '); iwrite (output,g); writeln;
  g:= 1/(1/f - 1/b);
  write ('g = 1 / (1/f - 1/b) = '); iwrite (output,g); writeln;
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 Ablaufprotkoll aufgezeigten Ergebnisse zeigen, daß die üblicherweise verwendete Methode zur Fehlerabschätzung ein falsches Intervall berechnet.

Aufgabe 4: 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 unter der Voraussetzung $f(a) \cdot f(b) < 0$ mit Hilfe des Intervall-Newton-Verfahrens

$$X_{n+1} := \left(m(X_n) - \frac{f(m(X_n))}{f'(X)} \right) \cap X_n$$

verbessert werden. Dabei ist $m(X)$ der Mittelpunkt von X.

Schreiben Sie unter Verwendung des Moduls IRARI ein PASCAL–SC 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 $X = [x_1, x_2]$ durch $m := x_1 + (x_2 - x_1)/2$ berechnet,
- ein Hauptprogramm, das das Startintervall X einliest, die zwei Kriterien $0 \in$ `F`(X) 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 Punktintervall eingeschlossen werden muß.

Lösung:

```
program i_newton (input,output);

use irari;

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:= X.inf + (X.sup - X.inf)/2;
  end;

begin
  write ('Startintervall: '); iread (input, Y);
  writeln;
  writeln ('Iteration'); writeln;
  if (0 in F(Y)) and not (0 in DF(Y)) then
    repeat
      iwrite (output,Y); writeln;
      X:= Y;
      Y:= ( m(X) - F ( intpt(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 5: 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

$$F(x, Y) = \begin{pmatrix} f_1(x, y_1, \ldots, y_n) \\ \vdots \\ f_n(x, y_1, \ldots, y_n) \end{pmatrix}.$$

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–SC Programm, das unter Verwendung des Moduls MVRARI im Bereich $0 \leq x \leq 1$ ausgehend von $Y(0) = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ die Werte von Y an den Stellen $x_i = i * h$, $i = 1, \ldots, 10$ 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 sich dazu die Vektorfunktion `F(x,Y)` und berechnen Sie dann in einer Schleife unter Verwendung der in MVRARI vordefinierten Operatoren jeweils die Ausdrücke `K1`, `K2`, `K3`, `K4` und den Wert von $Y(x_i)$.

Lösung:

```
program Runge (input,output);

use mvrari;

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      1.099554527492E+00   1.121329802764E-01   1.093200700450E+00
 0.2000      1.196656722926E+00   2.464289779975E-01   1.175471030585E+00
 0.3000      1.288616861810E+00   4.048255435744E-01   1.245988604002E+00
 0.4000      1.372263426908E+00   5.891563485406E-01   1.304048542331E+00
 0.5000      1.443906562486E+00   8.010876477500E-01   1.349070521045E+00
 0.6000      1.499304897507E+00   1.042044376333E+00   1.380604569959E+00
 0.7000      1.533637217111E+00   1.313125220557E+00   1.398335570436E+00
 0.8000      1.541480682053E+00   1.615006099199E+00   1.402086404355E+00
 0.9000      1.516797527292E+00   1.947831623546E+00   1.391819723375E+00
 1.0000      1.452932410994E+00   2.311094274287E+00   1.367638320819E+00
```

Bemerkung: In den Lösungen zu Aufgabe 4 und 5 wird deutlich, daß das allgemeine Operatorkonzept in PASCAL–SC die Übertragung von numerischen Algorithmen in Programmcode wesentlich vereinfacht. Im Prinzip lassen sich die mathematischen Formeln direkt als Anweisungszeilen übernehmen.

Aufgabe 6: Polynomauswertung

Mit Hilfe des Moduls *lss* 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 LGS

$$\begin{pmatrix} 1 & & & 0 & \\ -t & 1 & & & \\ & \ddots & \ddots & & \\ & & -t & 1 & \\ 0 & & & -t & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

bzw.

$$Ax = b$$

mit

$$A = (a_{ij}), \quad a_{ij} = \left\{ \begin{array}{ll} 1 & \text{für } i = j \\ -t & \text{für } i = j+1 \\ 0 & \text{sonst} \end{array} \right\}, \quad i, j = 0, \ldots, n$$

und

$$b = (b_i), \quad b_i = a_{n-i}, \quad i = 0, \ldots, n.$$

Schreiben Sie ein PASCAL–SC-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 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 Teil (b) die Polynomkoeffizienten $a_0, \ldots, a_n$ von p und die Auswertestelle t eingelesen werden,
- mit 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 abgearbeitet 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 die Module MVRARI und MVIRARI, in denen die dynamischen Typen *rvector*, *rmatrix* und *ivector* vereinbart sind, und das Modul LSS. 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 mvrari, mvirari, lss;

type  polynom = dynamic array [*] of real;

procedure read_poly (var p: polynom);
  var
    i: integer;
  begin
    for i:= lbound(p) to ubound(p) do
    begin
      write ('Koeff. ',i:2,': ');
      read (p[i]);
    end;
  end;

function horner (p : polynom; t: real) : real;
  var
    h: real;
    i: integer;
  begin
    h:= 0;
    for i:= ubound(p) downto lbound(p) 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_poly (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: '); iwrite (output, X[n]); writeln;
    end
    else
      writeln ('Error ',error:1,' aufgetreten');
  end;

var  n: integer;

begin
  write ('Polynomgrad: ');  read (n);
  main (n);
end.
```

Ablaufprotokoll:

```
Polynomgrad: 3
Polynom eingeben
Koeff.  0 =  1536796802
Koeff.  1 = -1086679440
Koeff.  2 = -768398401
Koeff.  3 =  543339720
Auswertestelle  t = 1.4142135

Hornerschema  : 1.000000000000E-03
Einschliessung: [5.978758735594E-06,5.978758735596E-06]


Polynomgrad: 3
Koeff.  0 =  170.4
Koeff.  1 = -356.41
Koeff.  2 =  168.97
Koeff.  3 =   18.601
Auswertestelle  t = 0.916079759
```

```
Hornerschema  : 0.000000000000E+00
Einschliessung: [1.148703133435E-13,1.148703133436E-13]
```

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 liefert.

Literatur

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

[2] Bohlender, G., Rall, L. B., Ullrich, Ch. and Wolff von Gudenberg, J.: *PASCAL–SC: A Microcomputer Language for Scientific Computation.* Academic Press, New York, 1986.

[3] Bohlender, G., Ullrich, Ch. and Wolff von Gudenberg, J.: *New Developments in PASCAL–SC.* In: SIGPLAN Notices, Vol. 23, **8**, 1988.

[4] Bohlender, G., Ullrich, Ch. and Wolff von Gudenberg, J.: *The Module Concepts in PASCAL–SC, Ada, Modula 2 and Fortran 8x.* In: Journal for Pascal, Ada & Modula 2, 1989.

[5] Kulisch, U.: *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik.* Bibliographisches Institut, Mannheim, 1976.

[6] Kulisch, U. (Ed.): *PASCAL–SC: A Pascal Extension for Scientific Computation*, Information Manual and Floppy Disks, Version IBM PC. B. G. Teubner, Stuttgart, John Wiley & Sons, Chichester, 1987.

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

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

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

[10] Kulisch, U. and Miranker, W. L.: *The Arithmetic of the Digital Computer: A New Approach.* SIAM Rev., 1-**40**, 1986.

[11] Neaga, M.: *PASCAL–SC – Eine PASCAL-Erweiterung für wissenschaftliches Rechnen.* In [8], 1989.

[12] Wolff von Gudenberg, J.: *Rechnerarithmetik für wissenschaftliches Rechnen - PASCAL–SC.* In: Höhler, G. und Staudenmaier, H. M. (Hrsg.): *Computer Theoretikum und Praktikum für Physiker*, **1**, 1985.

Dietmar Ratz
Institut für Angewandte Mathematik
Universität Karlsruhe
Kaiserstr. 12

7500 Karlsruhe 1