Zielgerichtete Algorithmenkonstruktion – von der Spezifikation zum Programm

Dr. Hans-Jürgen Buhl

1993

Fachbereich Mathematik (7)
Institut für Angewandte Informatik
Bergische Universität – Gesamthochschule Wuppertal

Interner Bericht der Integrierten Arbeitsgruppe
Mathematische Probleme aus dem Ingenieurbereich
IAGMPI - 9303
Mai 1993

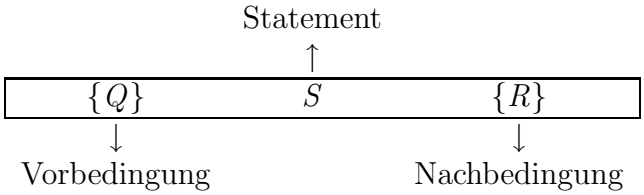
Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung: Mathematische Methoden und Theorie der Programmierung

1.1 Logische Argumentation – Spezifikation einzelner Anweisungen



(„Hoare triple“)

ist eine funktionale Spezifikation von S.
(Bedeutung: „Falls bei Gültigkeit von Q S ausgeführt wird, so ist R gültig.“)

Beispiele:

a) `var x,y,q,r : int;`
⋮
{x≥ 0 ∧ y>0}
S_1;
{q*y+r=x ∧ 0≤ r<y}
⋮

b) `var x,y : int;`
⋮
{x=X ∧ y=Y}
S_2;
{x=Y , y=X}
⋮

c) `var b : array of int;`
⋮
{#b > 0}
`var x : int;`
S_3;
{x=max{b[i]| 0≤i< #b}}
⋮

1.2 Verifikation – Korrektheitsbeweis

```
type
     Positive = 1..maxint;
```

```
function Power (x: real; i: Positive) : real;
     ⋮ [1]
var
     Akku : real;
begin
     {i=I0 ∈ [1,2,...maxint]}
     Akku := x;
     {Akku = Wert(x) =: X0 ∧ Akku * X0^(i-1) = X0^I0 ∧ i>0 }
     while (i>1) do
     begin
          Akku := Akku * x;
          { Akku * X0^(i-2) = X0^I0 }
          i:= pred(i);
          { Akku * X0^(i-1) = X0^I0 ∧ i>0 }
     end; {Terminierung nach (genau) I0-1 Schleifendurchgängen}
     { i=1 ∧ Akku = X0^I0 }
     Power := Akku
     { Power = X0^I0 }
end;
```

$***$ S vorhanden, Q vorhanden; beweise R

[1] Spezifikation erforderlich: wirklich im ganzen Definitionsbereich erfolgreiche Implementierung?

1.3 Spezifikation von Problemen/Systemen

Versuche Q und R für ein zu lösendes Problem zu beschreiben.

1.3.1 Das Parkplatzproblem

Beispiel:

„Informelle Beschreibung“: Auf einem Parkplatz stehen PKW's und Motorräder. Zusammen seien es n Fahrzeuge mit insgesammt m Rädern. Bestimme die Anzahl P der PKW's.

„Lösung“: Sei

P := Anzahl der PKW's
M := Anzahl der Motorräder

$$\left\{\begin{array}{c} P+M=n \\ 4P+2M=m \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{c} M=n-P \\ P=\frac{m-2n}{2} \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{c} M=\frac{4n-m}{2} \\ P=\frac{m-2n}{2} \end{array}\right\}$$

„Algorithmus“:

```
⋮
M := (4*n-m)/2;
P := (m-2*n)/2;
write (M,P);
⋮
```

Problem:
$***$ Null–Mark–Rechnung, Null–Mark–Mahnung,...
$(m,n)=(9,3) \Longrightarrow P=1\frac{1}{2}$
$(m,n)=(2,5) \Longrightarrow P=-4$

Vor der Entwicklung eines Algorithmus ist zunächst für das Problem eine funktionale Spezifikation, bestehend aus

1) Definitionsbereich,
2) Wertebereich und
3) für die Lösung wichtigen Eigenschaften (insbesondere funktionaler Zusammenhang zwischen Eingabe- und Ausgabegrößen)

anzufertigen!

bessere Beispiele:

a) **Eingabe:** $m, n \in \{0, 1, \ldots, \text{maxint}\}^2$

Vorbedingung: m gerade, $2n \leq m \leq 4n$

Ausgabe: $P \in \{0, 1, \ldots, \text{maxint}\}^2$, falls die Nachbedingung erfüllbar ist (sonst „keine Lösung“).

Nachbedingung: Ein $(P, M) \in \{0, 1, \ldots, \text{maxint}\}^2$ mit

$$\begin{array}{rcrcl} P & + & M & = & n \\ 4P & + & 2M & = & m \end{array}$$

b) „$\forall x \in \mathbb{R}$ berechne $\sin(x) \in \mathbb{R} \cap [-1.0, +1.0]$“ ???

1.3.2 $Ax^2 + Bx + C = 0$

Eingabe: $A, B, C \in$(IEEE_real $\setminus \{NaN, \pm\infty\}$), eventuell auch ohne $\{0\}$
Vorbedigung: true
Ausgabe:

a) IEEE FloatingPoint

Approximierte Lösungen von $Ax^2 + Bx + C = 0$ „gerundet“ in IEEE_real-Raster

Nachbedingung:

$\forall A \neq 0$**:** $X_i \in$ IEEE_real mit $\mid X_i - x_i \mid < n \cdot \varepsilon \cdot \mid x_i \mid$ mit $x_{1/2} \in \mathbb{C}$ als Lösungen von $Ax^2 + Bx + C = 0$, falls $x_i \in round^{-1}$ (IEEE_real $\setminus \{NaN, \pm\infty\}$) sonst ... [3] [4] (eventuell Realteil- und Imaginärteilweise).

$A = 0, \forall B \neq 0$**:** ...

$A = 0, B = 0, \forall C \neq 0$**:** „Keine Lösung“.

$A = 0, B = 0, C = 0$**:** „Jede Zahl ist Lösung“.

Bemerkung:

- Bei (in der Spezifikation) vorgegebenem n ist die Genauigkeitsforderung eventuell nur mit zusätzlichem Aufwand (mehrfache Genauigkeit,...) erreichbar!
- Wie überprüfe ich die Einhaltung der Genauigkeitsforderung/Verifikation (Wilkinsonsche Fehleranalyse, etc.)?

[2] Bitte $P \in \{0, 1, \ldots, \text{maxint}\}$, nicht $\mathbb{N}$ wie in der Literatur!
[3] ε=Maschinengenauigkeit, in IEEE_real gilt z.B.: $\varepsilon = 5.960E{-}8$ bzw. $\varepsilon = 1.110E{-}16$ (double)
[4] Zur Wahl von n vgl. Numerik!

Eigenschaften der Arithmetik: <float.h> aus **C**

The following describes floating-point representations that also mean the requirements for single-precision and double-precision normalized numbers in the *IEEE Standard for Binary Floating-Point Arithmetik* (ANSI IEEE Std 754-1985) and the appropriate values in a <float.h> header for types float and double.

- Exaktes Rechnen, dann Runden
  - to nearest (default) $\ldots 5 \rightarrow$ least significant bit = zero
  - directed

  bei $+, -, *, /$ und *sqrt*
- *Remainder*
- Konversion verschiedener Typen ineinander

$$x_f = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k}. \qquad -125 \leq e \leq +128$$

$$x_d = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k}. \qquad -1021 \leq e \leq +1024$$

```
FLT_RADIX                          2
FLT_MANT_DIG                      24
FLT_EPSILON             1.19209290E-07
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN                 1.17549435E-38
FLT_MIN_10_EXP                   -37
FLT_MAX_EXP                      128
FLT_MAX                 3.40282347E+38
FLT_MAX_10_EXP                    38
DBL_MANT_DIG                      53
DBL_EPSILON     2.2204460492503131E-16
DBL_DIG                           15
DBL_MIN_EXP                    -1021
DBL_MIN         2.225073858507201E-308
DBL_MIN_10_EXP                  -307
DBL_MAX_EXP                    +1024
DBL_MAX         1.797693134862316E+308
DBL_MAX_10_EXP                  +308
```

The values shown above for FLT_EPSILON and DBL_EPSILON are appropriate for the ANSI/IEEE Std 754-1985 default rounding mode (to nearest). Their values may differ for other rounding modes.

Problem in Standard-C: Nicht unterstützt werden:

- gradueller Underflow,

$$x_f = s \times 2^{-126} \times \sum_{k=2}^{24} f_k \times 2^{-k}$$

- NaN, infinity,
- Vergleiche (Probleme: nicht_vergleichbar),
- Traps/Ausnahmebedingungen und Trap-Handler (invalid Operation, DivBy-Zero, Overflow, Underflow (evtl. grad Underflow), Inexactness (rounding was necessary)

Kleinste (denormalisierte) Zahlen mit Signifikanzverlust:

FLT: 1.4013E–45
DBL: 4.9407E–324

Probleme:

- Obige Konstanten werden durch Wandlung String → real verfälscht. (Bitmustervergleich etwa bei: Turbo C 2.0/Atari ST). Vgl. nächster Abschnitt.
- $\pm 0, \sqrt{-0} = -0, \ldots$
- $\infty + \infty = \infty, -\infty \leq r \leq +\infty \quad \forall r \in$ IEEE_real $\setminus \{NaN\}$.
- { less,equal,greater,unordered} ← unordered, if NaN is at least one argument.
- signalisierende NaN's: nicht initialisierte Werte, $\overline{\mathbb{R}}$-Modell,...
- stille NaN's: Operationen auf NaN's, $(+\infty)+(-\infty), \ldots, 0/0, \infty/\infty$,sqrt$(-1.0)$, unvergleichbare Operanden

Empfohlene Funktionen und Prädikate:

(1) `Copysign(x,y)`, x mit Vorzeichen von y, d.h. `abs(x)=Copysign(x,1.0)`.

(2) $x \neq x - 0$ etwa bei: $x = -0$, NaN

(3) `Scalb(y,N)`$=y \cdot 2^N$ durch Shiften für ganzzahlige N.

(4) `Logb(x)`= unbiased exponent von x

bzw.

Logb(NaN)=NaN, Logb$(\infty)= +\infty$, Logb$(-\infty) = -\infty$

$0 <$Scalb$(x, -$Logb$(x))\begin{cases} < 2 \\ < 1 \end{cases}$, falls und nur falls denormalisiert

(5) `NextAfter(x,y)`= nächster Nachbar-Rasterpunkt von x in Richtung von y.

(6) `Finite(x)`

(7) `Isnan(x)` ($\Longleftrightarrow x \neq x$)

(8) x <> y ($\Longleftrightarrow x < y$ oder $x > y$)

Not($x = y$) ($\Longleftrightarrow x < y$ oder $x > y$ oder ein Operand ist NaN)

...

(9) `Unordered(x,y), x?y`

$$\begin{pmatrix} ? <> & \text{unordered or unequal,} \\ ? > & \text{unordered or greater,} \ldots \end{pmatrix}$$

(10) `Class(x)=`

- signaling NaN
- quiet NaN
- $-\infty$
- negative normalized non-zero
- negative denormalized non-zero
- -0
- $+0$
- positive denormalized non-zero
- positive normalized non-zero
- $+\infty$

- IEEE REAL format: (1 + 8 + 23 bit = 32 bit)

	s	e	f
Bit	31	30..23	22..0

Exponent = $e - 126$; e = Exponent + 126; 126 = 7EH.

Mantissa = 1f.

e (8 bit) 00H..FFH = 0..255	f (23 bit)	value:
FFH	<> 0	NaN
FFH	= 0	(-1)^s * infinity
01H..FEH (normalized)		(-1)^s * (0.1f) * 2^(e-126).
00H	<> 0	(-1)^s * (0.f) * 2^(-126).
00H	= 0	(-1)^s * 0.0

- IEEE DOUBLE format: (1 + 11 + 52 bits = 64 bits)

	s	e	f
Bit	63	62..52	51..0

Exponent = $e - 1022$; e = Exponent + 1022; 1022 = 3FEH.

Mantissa = 1f.

e (11 bit) 000H..7FFH = 0..2047	f (52 bit)	value:
7FFH	<> 0	NaN
7FFH	= 0	(-1)^s * infinity
001H..7FEH (normalized)		(-1)^s * (0.1f) * 2^(e-1022).
000H	<> 0	(-1)^s * (0.f) * 2^(-1022).
000H	= 0	(-1)^s * 0.0

b) Intervallrechnung/gerichtete Rundung

Intervalle $X_i^{Intervall}$ mit Endpunkten in IEEE_real, so daß X_1, X_2 garantiert in den angegebenen Intervallen liegen.

Nachbedingung: $\underbrace{d(X_i^{\text{Intervall}})}_{\text{Durchmesser}} < n \cdot \varepsilon \cdot \underbrace{|\, X_i^{\text{Intervall}} \,|}_{\text{betragsmaximaler Wert}}$

Bemerkung: Durch naive Intervallversion/gerichtete Rundung des „üblichen“ Algorithmus evtl. leicht erreichbar, wenn n nicht „zu klein“ vorgegeben ist;
alternativ vgl. c)

c) Einschließung

(Einschließung = „Verfikation“ im Sinne der Intervallrechnung $\neq$ Verifikation im Sinne der Algorithmentheorie; Alternative zu a) und b))

Ausgabe: wie bei b)

Nachbedingung: wie bei b)

Bemerkung:

- Berechne irgendwie eine Näherungslösung (etwa a)) und benutze anschließend einen Fixpunktsatz zur Einschließung und evtl. nötigen iterativen Verbesserung des Durchmessers der Einschließung. (vgl. Vorlesung „Konstruktives numerisches Rechnen“ (Prof. Heindl))
- Diese Spezifikation ist technisch komplex, verhüllt aber nicht die Ungenauigkeit!
- „Verifizieren“ und „zielgerichtetes Konstruieren“ ist nur bei Benutzung genauer spezifizierter arithmetischer Grundoperationen möglich!

1.3.3 readreal und writereal / Fallstudien

Wandle einen String in einen reellen Wert

Eine in Stringform gegebene „reelle Zahl“ in eine (interne) „real“-Zahl umzuwandeln ist nicht so einfach, wie in der (idealen) Mathematik beschrieben:

$\underbrace{''b_n b_{n\text{-}1} \cdots b_1 b_0 \cdot b_{\text{-}1} b_{\text{-}2} \cdots b_{-m}\,''}\quad n \in \mathbb{N}, m \in \mathbb{N}$

```
var
     b              : array of character;      {Input}
     a              : real;                    {Result}
     ch             : character;
     n,m            : integer;
     vork           : boolean;

     a := 0.0;
     vork := true;
     ch := getchar(b);
     n := -1; m := 0;

while (ch <> EOF) do begin
     if (ch in ['0',..,'9']) then begin
          a := a*10.0+(ord(ch)-ord('0'));
          if (vork) then
               n := succ(n)
          else
               m := pred(m);
     end else if (ch = '.') then
          vork := false
     else
          "Fehlerbehandlung";
     ch := getchar(b);
end; {while}
a := a / Power(10.0,m);
```

Hier wird nämlich davon ausgegangen, daß die Beziehung

$$10.0/10.0 = 1.0$$

gilt: Darüber hinaus wurde nicht explizit auf die (gewünschte) Rundung ins real-Format geachtet, wenn die Dezimalzahl „b“ nicht exakt in real darstellbar ist (5^{-n}, zu viele Mantissenstellen). Außerdem kann es bei Zulassung von „nichtnormalisierten“ Strings (Exponentenformat) bei mathematisch equivalenten Zahlen

$$1.2001 \cdot 10^{-38}$$
$$0.012001 \cdot 10^{-36}, \ldots$$

zu Problemen (unnötiger Zwischendurch-Underflow/Overflow, denormalisierte Zwischenergebnisse, unnötige $*10.0/10.0$-Operationspärchen,...) kommen. Eine Musterspezifikation, die auf das vorhandene Gleitkommasystem Rücksicht nimmt, ist zum Beispiel die folgende:

```
{=========================================================================}
{ File: STRTOR.DOC                  Date of project plan: July, 12th., 1987 }
{                                                                           }
{ Author: (c) Bergische Universitaet Wuppertal, HJB                         }
{         PUBLIC DOMAIN, if this line of source code isn't omitted.         }
{                                                                           }
{ History:  Version 1       July, 13th, 1987         main implementation    }
{                                                                           }
{=========================================================================}
{                                                                           }
{          +-----------------------------------------------------------------}
{          | function StringToReal(    NumberASCII : string;                }
{          |                         var NumberReal  : real  ): SuccessStates;}
{          +-----------------------------------------------------------------}
{          | type                                                            }
{          |            ScuccessStates = (     OK,                           }
{          |                                 PartOK,                         }
{          |                                  NotOK );                       }
{          +-----------------------------------------------------------------}
{          |                                                                 }
{          |   This function converts a real number given in its external    }
{          |   representation, i.e. in the form                              }
{          |                                                                 }
(*         |       +-------------------------------------------------+   *)
(*         |       | {" "} ["+"|"-"] digit {digit} ["." digit {digit}] |   *)
(*         |       |       [("E"|"e") ["+"|"-"] digit {digit}]         |   *)
(*         |       +-------------------------------------------------+   *)
{          |                                                                 }
{          |   into its internal "representation" (machine number system -   }
{          |   see implementation specific description of the INTERNAL       }
{          |   FLOATING-POINT REPRESENTATION) and signals the state of       }
{          |   success of the conversation:                                  }
{          |                                                                 }
{          |         OK,  if the string "NumberASCII" contains a syntacti-   }
{          |              cally correct number with an absolute value        }
{          |              not beyond the largest machine number;             }
{          |     PartOK,  if the string "NumberASCII" is made of two parts,  }
{          |              the first containing a syntactically correct       }
{          |              real number with an absolute value not beyond      }
{          |              the largest machine number and the second starting}
{          |              with the blank character " ";                      }
{          |      NotOK,  otherwise.                                         }
{          |                                                                 }
{          |   The value of "NumberReal" will only be changed, if the        }
```

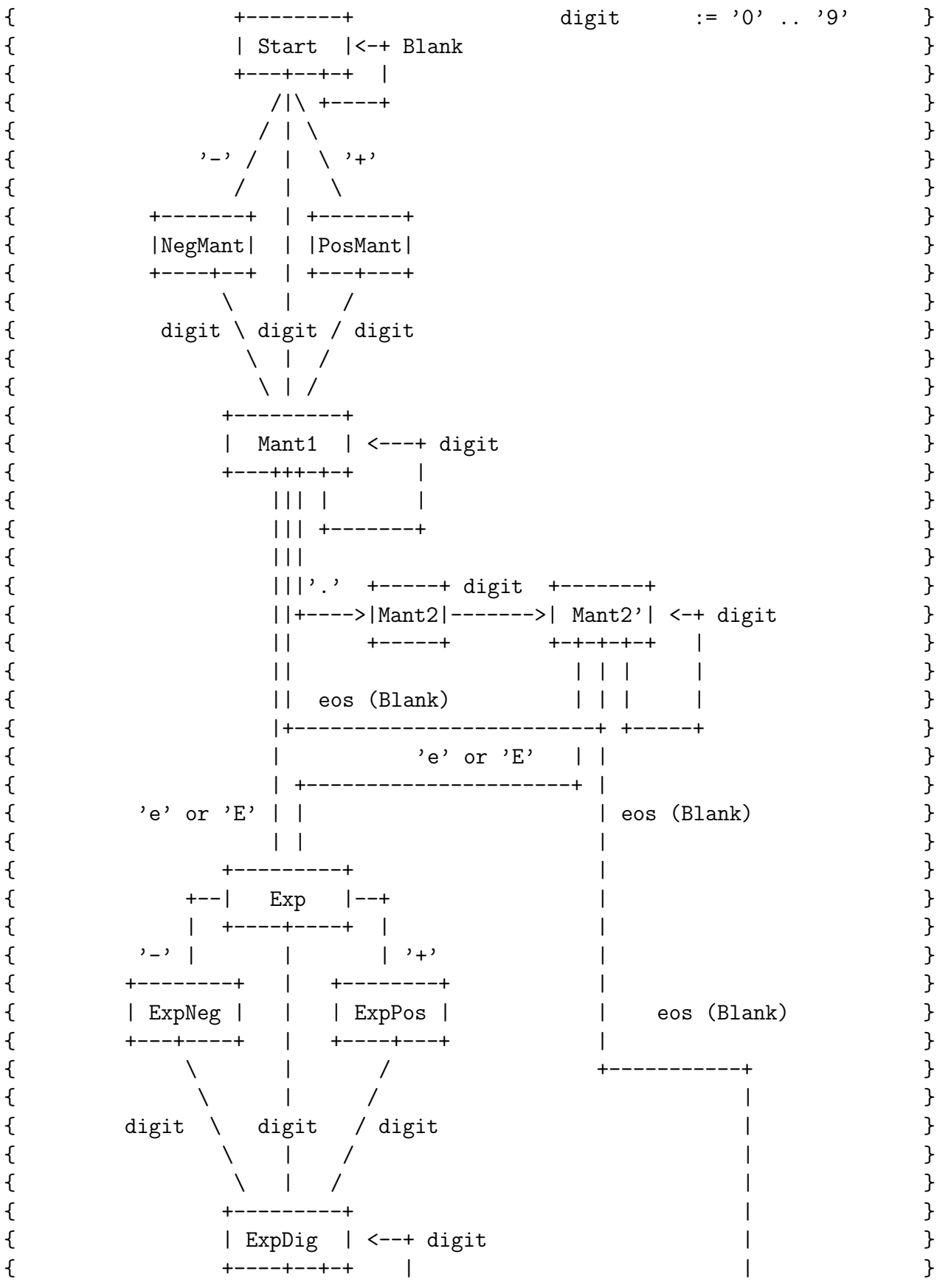
```
{          |   conversion ends successfully (i.e. not in state "NotOK")!    }
{          |       (<^C> halts the program calling "StringToReal"!)         }
{          +-----------------------------------------------------------------}
{                                                                            }
{          +-----------------------------------------------------------------}
{          | Usage:                                                         }
{          |                                                                }
{          |   type  SuccessStates = ( OK, PartOK, NotOK );                 }
{          |                                                                }
{          |   var                                                          }
{          |         InputLine     :  string;                               }
{          |         InputVar      :  real;                                 }
{          |                                                                }
{          |   {$I STRTOR.INC}                                              }
{          |                                                                }
{          |         ...                                                    }
{          |         repeat                                                 }
{          |             GotoXY(5, 15); ClearEndOfLine;                     }
{          |             write(' ..promptstring.. ');                       }
{          |             readln(InputLine);                                 }
{          |         until (StringToReal(InputLine, InputVar) <> NotOK);    }
{          |         ...                                                    }
{          |                                                                }
{          +-----------------------------------------------------------------}
{                                                                            }
{                                                                            }
{       Conditions to be fulfilled:                                          }
{       ---------------------------                                          }
{                                                                            }
{       -  The conversion result should be as close as possible to the       }
{          given (external) number - "rounding to raster of machine          }
{          numbers" ( numbers less than "smallReal" convert to 0.0 ).        }
{                                                                            }
{       -  If the (external) number is to large to be represented in the     }
{          internal number system the result of the conversion should be     }
{          "NotOK" - "floating point overflow".                              }
{                                                                            }
{       -  All natural numbers exactly representable in the machine number   }
{          raster should be converted to their exact internal                }
{          representation!                                                   }
{                                                                            }
{       Preconditions, the algorithm depends on:                             }
{       ----------------------------------------                             }
{                                                                            }
{       -  All the numbers 0..9 and 10 have an exact internal                }
```

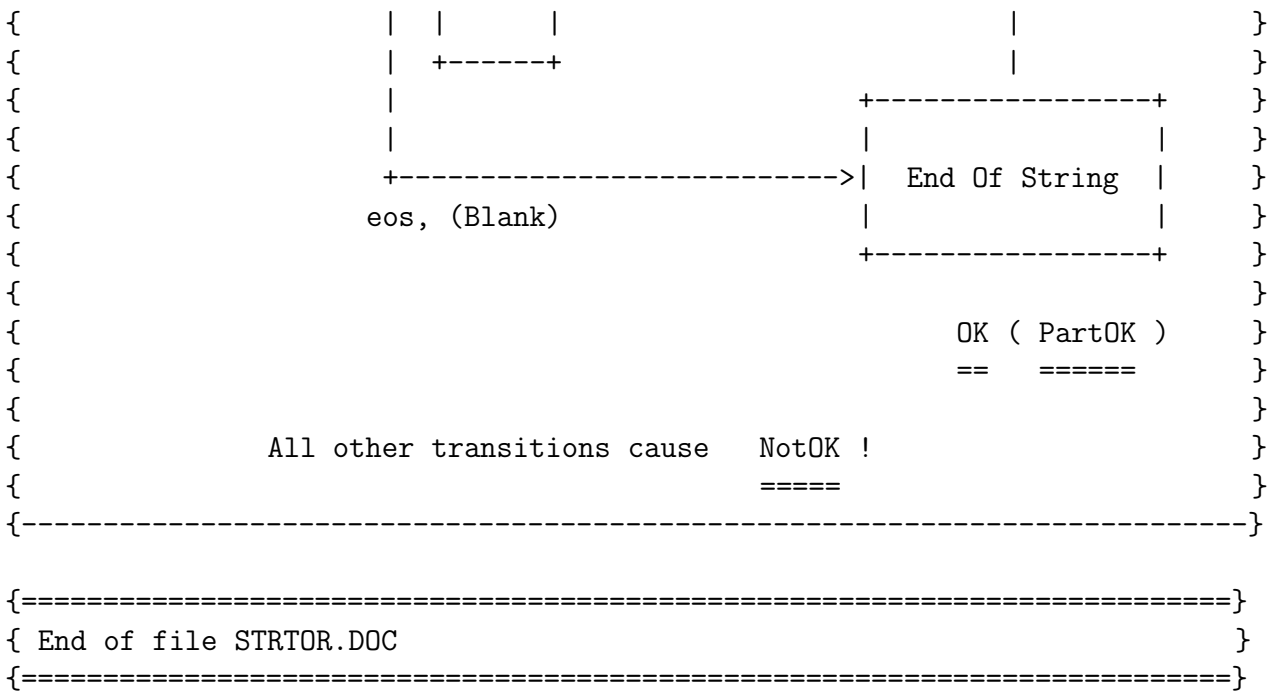
```
{          representation in the internal floating point system.            }
{                                                                           }
{       -  The result of adding a number 0..9 to an exactly represented     }
{          number is also exactly represented whenever there exists such    }
{          a representation.                                                }
{                                                                           }
{       -  The result of multiplying an exactly represented number by 10    }
{          is also exactly represented whenever the product is exactly      }
{          representable.                                                   }
{                                                                           }
{                                                                           }
{       Glossary:                                                           }
{       ---------                                                           }
{                                                                           }
{       external representation,                                            }
{          ~     number system        = (ASCII-)character representation    }
{                                       of a number in a decimal floating   }
{                                       point system, syntax see            }
{                                       definition above.                   }
{                                                                           }
{                                                                           }
{       internal representation,                                            }
{          ~     number system,                                             }
{          ~  floating point system,                                        }
{       machine number system        =  the form of representation          }
{                                       of real numbers, the computer       }
{                                       needs for its internal built in     }
{                                       arithmetic.                         }
{                                                                           }
{                                                                           }

{Pascal ST+   Pascal ST+   Pascal ST+   Pascal St+   Pascal ST+   Pascal ST+}
{                                                                       ST+}
{                                                                       ST+}
{       IMPLEMENTATION SPECIFIC INTERNAL FLOATING-POINT REPRESENTATION: ST+}
{       ============================================================== ST+}
{                                                                       ST+}
{       Type:   real                          Size:   6 bytes           ST+}
{                                                                       ST+}
{                                                                       ST+}
{       +-----+------------+-----------+      Base:              2     ST+}
{       | r47 | r46 ... r8  | r7 ... r0 |                               ST+}
{       +-----+------------+-----------+      number of                 ST+}
{       |  s  | m1  ... m39 | e7 ... e0 |      mantissa digits:  40     ST+}
{       +-----+------------+-----------+                               ST+}
```

```
{                                                 number of                ST+}
{       e = e7 .. e1 B (unsigned).                exponent digits:   8     ST+}
{                                                                          ST+}
{     +-------------------------------------------------------------+      ST+}
{     |                    +-    0.0,          if e = 0;             |     ST+}
{     |                    |                                        |     ST+}
{     |                    |                                        |     ST+}
{     |                    |                                        |     ST+}
{     | value(r47..r0)  = -+         s                    (e-80H)  |     ST+}
{     |                    |     (-1) * 0.1 m  m ...m   * 10       , |     ST+}
{     |                    |                 1  2    39             |     ST+}
{     |                    |                                        |     ST+}
{     |                    +-                  if e <> 0.           |     ST+}
{     +-------------------------------------------------------------+      ST+}
{                                                                          ST+}
{                                                                          ST+}
{                                                                          ST+}
{      minExpReal =   1-128 = -127 (= 0FF81H)                              ST+}
{      maxExpReal = 255-128 = +127 (= 0007FH)                              ST+}
{                                                                          ST+}
{                                                                          ST+}
{      RoundingReal = TRUE,  EpsilonReal = 0.5 * 2^(1-40) = 9.095E-13      ST+}
{                                                                          ST+}
{      maxReal    = (1.0-2^(-41)) * 2^127 =  1.701411835E38,               ST+}
{      ln(maxReal)                        =  88.02969193,                  ST+}
{                   [ 7F FF FF FF FF FF  H ]                               ST+}
{                                                                          ST+}
{                                                                          ST+}
{      smallReal  = 0.5 * 2^(-127)        =  2.938735877E-39,              ST+}
{      ln(smallReal)                      = -88.72283911,                  ST+}
{                   [ 00 00 00 00 00 01  H ]                               ST+}
{                                                                          ST+}
{                                                                          ST+}
{            ( NULL  =  ?? ?? ?? ?? ?? 00  H )                            ST+}
{                                                                          ST+}
{                                                                          ST+}
{Pascal ST+   Pascal ST+   Pascal ST+   Pascal St+   Pascal ST+   Pascal ST+}

{------------------------------------------------------------------------}
{                                                                         }
{      Transition states analyzing the syntax of a real number:           }
{      ---------------------------------------------------------          }
{                                                                         }
{                                          Blank      := ' ';             }
{                                                                         }
```


digit := '0' .. '9'
Start
Blank
'-'
'+'
NegMant
PosMant
digit
digit
digit
Mant1
digit
'.'
digit
Mant2
Mant2'
digit
eos (Blank)
'e' or 'E'
'e' or 'E'
eos (Blank)
Exp
'-'
'+'
ExpNeg
ExpPos
eos (Blank)
digit
digit
digit
ExpDig
digit



{==}
{ End of file STRTOR.DOC   }
{==}

Die folgenden zwei Prozeduren des PASCAL 6000 Systems zeigen die Umwandlung reeller Zahlen von ihrer dezimalen Darstellung in eine beliebige „interne“ Darstellung und umgekehrt. (Die Konstanten im Prozedur-Kopf sind bestimmt durch die Eigenschaft des Formates der Gleitkommazahlen der Rechenanlage CDC 6000: 11 Bit binärer Exponent und 48 Bit Mantisse.)

```
procedure readreal(var f: text; var x:real);
   {Einlesen einer reellen Zahl x vom File f}
   {folgende Konstanten sind systemabhängig }
   const t48 = 281474976710656;      {48 Bit Mantisse}
         limit = 56294995342131;     {= t48 div 5}
         z = 27;                     {= ord('0') }
         lim1 = 322;                 {grösster Exponent}
         lim2 = -292;                {kleinster Exponent}
   type posint = 0..323;
   var ch: char; y: real; a,i,e: integer;
       s,ss: boolean;                {Vorzeichen}

   function ten(e: posint):real;     { = 10**e, 0<e<322 }
      var i: integer; t: real;
      begin i := 0; t := 1.0;
         repeat if odd(e) then
            case i of
             0: t := t * 1.0e1;
             1: t := t * 1.0e2;
```

```
          2: t := t * 1.0e4;
          3: t := t * 1.0e8;
          4: t := t * 1.0e16;
          5: t := t * 1.0e32;
          6: t := t * 1.0e64;
          7: t := t * 1.0e128;
          8: t := t * 1.0e256;
        end ;
        e := e div 2; i := i+1;
    until e = 0;
    ten := t
  end ;

begin
  if eof(f) then
  begin message(' tried to read past end of file f');
        halt
  end;
  {überspringe führende Leerzeichen}
  while (f^ = ' ') and  (not eof(f)) do get(f);
  if not eof(f) then
  begin
    ch := f^;
    if ch = '-' then
      begin s := true; get (f); ch := f^;
      end else
      begin s := false;
        if ch = '+' then
        begin get (f); ch := f^
        end
      end ;
    if not (ch in ['0'..'9']) then
    begin message('  digit expected '); halt;
    end;
    a := 0; e := 0;
    repeat if a < limit then a := 10*a + ord(ch)-z
                        else e := e+1;
      get(f); ch := f^
    until not (ch in ['0'..'9']);
    if ch = '.' then
    begin { Einlesen des Bruches } get(f); ch := f^;
      while ch in ['0'..'9'] do
      begin if a < limit then
            begin a := 10*a + ord(ch)-z; e := e-1
            end ;
```

```
            get(f); ch := f^
         end
      end ;
      if ch = 'e' then
      begin { Einlesen des Skalierungsfaktors }
         i := 0; get(f); ch := f^;
         if ch = '-' then
         begin ss := true; get(f); ch := f^
         end else
         begin ss := false; if ch = '+' then
            begin get(f); ch := f^
            end
         end ;
         if ch in ['0'..'9'] then
         begin i := ord(ch)-z; get(f); ch := f^;
            while ch in ['0'..'9'] do
            begin if i < limit then i := 10*i + ord(ch)-z;
               get(f); ch := f^
            end
         end else
         begin message('  digit expected '); halt
         end ;
         if ss then e := e-i else e := e+i;
      end ;
      if e < lim2 then
         begin a := 0; e := 0
         end else
      if e > lim1 then
      begin message(' number too large '); halt end;
      { 0 < a < 2**49 }

      if a >= t48 then y := ((a+1) div 2) * 2.0 else y := a;
      if s then y := -y;
      if e < 0 then x := y/ten(-e) else
      if e <> 0 then x := y *ten(e) else x := y;
   end;
end  { readreal }
```

Wandle real-Wert in String:

```
procedure writereal(var f: text; x: real; n: integer);
   {Schreiben einer reellen Zahl x mit n Zeichen in dezimalem
```

```
 Gleitkommaformat}
{Die folgenden Konstanten hängen von der zugrundeliegenden
 Darstellung der Gleitkommazahl ab}

const t48 = 281474976710656; ⟵ hardcoded
        { = 2**48; 48 = Grösse der Mantisse }
        z = 27;        { ord('0') } ⟵ Std.-Pascal als Const nicht vermeidbar
type posint = 0..323; ⟵ hardcoded
                      ⟵ Abhängigkeit von der Implementation von „real“!
                          (→ evtl. Include-Datei, float.h,...)
{ Bereich des dezimalen Exponents }
var c,d,e,e0,e1,e2,i: integer; ⟵ Kommentar?

function ten(e: posint):real;   { = 10**e, 0<e<322 }
   var i: integer; t: real;
begin i := 0; t := 1.0;
   repeat if odd(e) then
      case i of
       0: t := t * 1.0e1;
       1: t := t * 1.0e2;
       2: t := t * 1.0e4;
       3: t := t * 1.0e8;
       4: t := t * 1.0e16;
       5: t := t * 1.0e32;
       6: t := t * 1.0e64;
       7: t := t * 1.0e128;
       8: t := t * 1.0e256;
      end ;
      e := e div 2; i := i+1;
   until e = 0;
   ten := t

end { ten } ; ⟵ Table-Lookup verbunden mit bitweisem
                 Abarbeiten des Exponenten im Dualsystem:
```

$$10^{11} = 10^{1011_B} = \prod_{\substack{b_i \neq 0}} 10^{2^i} = 10^1 \cdot 10^2 \cdot 10^8$$

$$\uparrow$$

$$1011_B = \begin{array}{lll} 2^0+ & 2^1+ & 2^3 \\ \uparrow & \uparrow & \uparrow \\ i=0 & i=1 & i=3 \end{array}$$

$$10^{\sum_{i=0}^{N} b_i \cdot 2^i} = \prod_{i=0}^{N} 10^{b_i \cdot 2^i} = \prod_{i=0}^{N} (10^{2^i})^{b_i} = \prod_{\substack{i=0 \\ b_i \neq 0}}^{N} 10^{2^i}$$

```
begin { es werden zumindest 10 Zeichen benötigt: b+9.9e+999 }
   if x = 0 then
   begin repeat write(f, ' '; n := n-1
         until n <= 1;
      write(f,'0')

   end else ⟵ führende Blanks; 0.0 als 0 ausgeben!
   { x <> 0 }
   begin
      if n <= 10 then n := 3 ⟵ Druckfeldadaption (Spezifikation:...)
                 else n := n-7;
      repeat write(f, ' '); n := n-1;
      until n <= 15; ⟵ abhängig vom Zahlenformat, hier hardcoded (schlecht!)
      { 1 < n <= 15, Zahl der zu druckenden Ziffern } ←führende Blanks


      begin { teste Vorzeichen und ermittle Exponent }
         if x < 0 then
            begin write(f, '-'); x := -x
            end else write(f, ' ');

         e := expo(x);  { e = entier(log2(abs(x))) }
               ↑ häufig unumgänglich, aber schlecht, vgl.
                 Logb(x) nach IEEE 754-Empfehlung!
         if e >= 0 then
            begin e := e*77 div 256 +1; ⟵Exponent bzgl Basis 10
                                          (→ x.yyyEnnn)


               x := x/ten(e);
               if x >= 1.0 then
                  begin x := x/10.0; e := e+1
                  end
            end else
            begin e := (e+1)*77 div 256; x := ten(-e)*x;
               if x < 0.1 then

                  begin x := 10.0*x; ←schlecht,vgl. IEEE 754-Empfehlung:
                                       Scalb(x, –Logb(x))

                     e := e-1
                  end
            end ;
         { 0.1 <= x < 1.0 }

         case n of   { Runden }
           2: x := x+0.5e-2;
```

```
          3: x := x+0.5e-3;
          4: x := x+0.5e-4;
          5: x := x+0.5e-5;
          6: x := x+0.5e-6;
          7: x := x+0.5e-7;
          8: x := x+0.5e-8;
          9: x := x+0.5e-9;
         10: x := x+0.5e-10;
         11: x := x+0.5e-11;
         12: x := x+0.5e-12;
         13: x := x+0.5e-13;
         14: x := x+0.5e-14;
         15: x := x+0.5e-15
       end; ⟵ Rundungsvorbereitung

       if x >= 1.0 then
          begin x := x * 0.1; e := e+1;
          end;

       c := trunc(x,48); ⟵hardcoded (≟(?)! trunc (x ∗ 2^48))
       c := 10*c; d := c div t48; ⟵ Herausschieben von Ziffern nach
                                     oben! (Integer-Arithmetik)
                                     Welche Arithmetik-Anforderungen?
       write(f, chr(d+z), '.');
                  ↑ Ziffernisolation, Vorkommaziffer + Komma

       for i := 2 to n do
       begin c := (c - d*t48) * 10; d := c div t48;
          write(f, chr(d+z))

       end; ⟵ Nachkommaziffern

       write(f, 'e'); e := e-1;
       if e < 0 then
          begin write(f, '-'); e := -e
          end else write(f, '+');

       e1 := e * 205 div 2048; ⟵ ≙ 0.1 in Integer-Arithmetik
       e2 := e - 10*e1;
       e0 := e1 * 205 div 2048; ⟵ ≙ 0.1 in Integer-Arithmetik
       e1 := e1 - 10*e0;
       write(f, chr(e0+z), chr(e1+z), chr(e2+z))
                               ↑ Ziffernextraktion in Integer-Arithmetik

    end
  end
end { writereal };
```

(Wie sieht die Spezifikation von „writereal“ aus? $\Longrightarrow$ Übungen)

1.3.4 Vermeidung von Overflows

– gute Kommentare, Namenswahl.

– „Spezifikation“ von Variablen so genau wie in der Programmiersprache möglich.

```
const
      ErrorResult = -1;
 ⋮

function fakultaet(n : integer {n >= 0}): integer;
var
        zaehl : 2..maxint;
        teilres : 1..maxint;   {Teilresultat}
begin
        teilres := 1;
        for zaehl := 2 to n do begin
              teilres := teilres * zaehl
        end;
        if (n >= 0) then fakultaet := teilres
                    else fakultaet := ErrorResult;
end;
```

– Verifikation:

Endlichkeit: Die Schleife wird genau $(n-1)$-mal durchlaufen, falls $n \geq 2$. Ansonsten wird sie kein mal durchlaufen.

Schleifenvariante: Zerlegung des gewünschten Ergebnisses $\boxed{n!}$ in ein schon berechnetes Teilergebnis `teilres` und einen noch zu berechnenden Rest:

$$\boxed{n! \;=\; \texttt{teilres} \;*\; \prod_{i=zaehl}^{n} i} \quad \text{bei Schleifenbegin.}$$

Korrektheit bei Abbruch:

```
begin
            {n in integer; zaehl, teilres undef}
    teilres := 1;
            {n! = teilres * n!}
    for zaehl := 2 to n do begin      {n >= 2}
            {n! = teilres * ∏_{i=zaehl}^{n} i, n ≥ zaehl}
        teilres := teilres * zaehl
            {n! = teilres * ∏_{i=zaehl+1}^{n} i, n ≥ zaehl}
    end;
            {n ≥ 2: n!=teilres·1, zaehl undef}

    {n ≥ 2: n! = teilres; n in [0,1]:teilres = 1 = n!}
    {n < 0: n! undef }
      ⋮
```

– informelle Spezifikation:

Zweck: „`fakultaet`“ berechnet die Fakultät von n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n,$$
$$0! = 1$$

für alle n in [0,1,...,maxint], für die $n!$ auch in `INTEGER` liegt.

Interface:

n in Argument, dessen Fakultät berechnet werden soll.

Result out Fakultät von n oder Fehlerkennzeichnung „ErrorResult“

Result $\in$ [ErrorResult]$\cup$[1,...,maxint].

Vorbedingungen:

n in integer.

Nachbedingung:

`fakultaet` liefert:

$n!$, falls $n \geq 0$ und $n! \leq$ maxint;

ErrorResult, falls $n < 0$;

undefiniertes Verhalten bei $n! >$ maxint.

Beachte: Diese Spezifikation ist für die Praxis ungeeignet! (LISP-Problem auf der Cyber,...). „Verhalten undefiniert“ hängt von Pascal und der Laufzeitumgebung ab (Overflow-Exception ja/nein?).

Da bei den meisten Pascal-Dialekten auf Microcomputern kein `INTEGER`-Overflow „erkannt“ wird, dann also durch obige Funktion ein falsches Resultat geliefert wird, sollte die Funktion so abgeändert werden, daß gilt:

`fakultaet(n)`$= n!$ <u>oder</u> ErrorResult

für alle n <u>in</u> `INTEGER`! („n <u>in</u> `INTEGER`“ ist die als Kommentar auch in Programmquellen mögliche Schreibweise von „$n \in$ `INTEGER`“.)

Problem: Bei der Verifikation wurde in $(*, \mathbb{Z})$ und nicht im Zahlenformat $(\odot,$ `INTEGER`$)$ gearbeitet!

– mögliche Varianten / Verbesserungen:

Ziel: Berechne `teilres := teilres * zaehl` nur dann, wenn das innerhalb [1,. . . ,maxint] möglich ist.

a)

⋮

```
if (maxint div teilres >= zaehl) then
     teilres := teilres * zaehl
else begin
     fakultaet := ErrorResult;
     goto 999 {end of fakultaet}
end;
```

b)

⋮

{n!=teilres$*\prod_{i=zaehl}^{n} i,$ n $\geq$ zaehl,}
{teilres <= maxint }
<u>if (maxint div teilres >= zaehl) then</u>
{teilres * zaehl <= maxint}
<u>teilres := teilres * zaehl</u>
{n!=teilres$*\prod_{i=zaehl+1}^{n} i,$ n $\geq$ zaehl,}
<u>else begin</u>
{n! $\geq$ teilres * zaehl $>$ maxint}
⋮

c)

⋮

{ Nachbedingungen: . . . <u>n!</u>, falls $n \geq 0$ <u>und</u> $n! \leq$maxint }
{ <u>ErrorResult</u> sonst }

Ziel: Modifikation von „`fakultaet`“ im Sinne des „dynamic programming“. („Merken“, statt immer wieder ausrechnen!) Sei maxArgument := $\max\{i \in \mathbb{N} \mid i! \leq \text{maxint}\}$.

```
⋮
const
     ErrorResult = -1;
     maxArgument = ...;

---

type
     CARDINAL = 0..maxint;

---

var
     FakWerte : ARRAY[0..maxArgument] of CARDINAL;
     FakObereGrenze : 0..maxArgument;
          {FakWerte [0..FakObereGrenze] bereits bekannt, }
          {     anfangs: 0! bekannt (Vorbesetzung)        }
⋮
```

```
function fakultaet(n : integer {n ≥ 0}) : integer;
label 999;
var
     zaehl : CARDINAL;
begin
     for zaehl := (FakObereGrenze+1) to n do
          if(maxint div FakWerte[zaehl-1] >= zaehl)
          then
               FakWerte[zaehl] := FakWerte[zaehl-1]*zaehl
          else begin
               .....
          end {if};
     fakultaet := FakWerte[n];
     FakObereGrenze := max(FakObereGrenze, n);
999 : end {function fakultaet};
```

```
⋮
begin
     FakObereGrenze := 0;
     FakWerte[0] := 1;
     ⋮
```

Ähnlich: Etwa bei einer verbesserten Version von „Power“ unter Zugrundelegung von

$$x^{2i} = (x^2)^i,\ x^{2i+1} = x^{2i} \cdot x$$

statt von

$$x^i = x^{i\text{-}1} \cdot x.$$

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do
          if odd(n) then begin
               Teilerg := Teilerg * x; n := n - 1;
          end else begin
               x := sqr(x); n := n div 2;
          end;
          ⋮
```

Folgende Implementierung versucht einige Fallunterscheidungen zu vermeiden:

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do begin
          while not odd(n) do begin
               x := sqr(x); n := n div 2;
          end;                                   { n ungerade}
          Teilerg := Teilerg * x; n := n - 1;
     end;
     ⋮
```

Im Algorithmus nach *O.J. Dahl/E.W. Dijkstra/C.A.R. Hoare: Structured Programming, Seite 14* wird eine solche Vermeidung auf andere Weise angestrebt:

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do begin
          if odd(n) then begin
               Teilerg := Teilerg * x; n := n - 1;   ← siehe 2 b)
          end;                                        { n gerade}
          x := sqr(x); n := n div 2;   ← siehe 1) und 2 a)
     end;
     ⋮
```

Obwohl diese drei Varianten in $(\mathbb{R},\ \mathbb{Z}_0^+)$ alle das richtige Ergbnis x^n lieferten, wird in der dritten Variante am Schluß eine unnötige Quadrierung von x vorgenommen (while-Statement mit $n = 1$). Das kann zu einem Overflow führen, obwohl der durch das entprechende `x := sqr(x)` berechnete x-Wert für die gesuchte Potenz garnicht mehr benötigt wird!
Es wird also für Variante 3 eine strengere Vorbedingung als nötig und wünschenswert gefordert (d.h. der Bereich der Argumente, für die x^n berechnet wird, ist unnötigerweise verkleinert worden):

- **Vorbedingung zu Variante 1 und 2:** x^n ist (im Rahmen der begrenzten Rechengenauigkeit in IEEE_real - vgl. etwa Abschnitt 3.1.2) kleiner oder gleich `FLT_MAX`.

- **Vorbedingung zu Variante 3:** x^n und $x^{2^{\lfloor log_2(n) \rfloor + 1}}$ ist (im Rahmen der begrenzten Rechengenauigkeit in IEEE_real - vgl. etwa Abschnitt 3.1.2) kleiner oder gleich `FLT_MAX`.

Es muß also für die Variante 3 nicht nur x^n, sondern auch die Potenz von x zum Exponenten der zu n nächsthöheren Zweierpotenz kleiner oder gleich `FLT_MAX` sein.
In zur Version 3 ähnlichen Fällen, bei denen nicht sofort zu einem besseren Algorithmus – wie hier Version 2 – ohne die Problematik des unnötigen Overflows übergegangen werden kann, ist ein zum folgenden analoges Vorgehen sinnvoll:

1) „Vermeide unnötigen Overflow in der Zwischenrechnung:“

```
n := n div 2;
if (n <> 0) then x := sqr(x)
```

2) „Fange unvermeidlichen Overflow bei der Berechnung ab:“

a) „... bei der Berechnung von `sqr(x)`:“

```
n := n div 2;
if (n <> 0) then
     if (x < SQR_FLT_MAX) then
                    ↑ selbst zu definierende Konstante aus
                      FLT_MAX (flaot.h,etc.)
               x := sqr(x)
     else ...;
```

b) „... bei der Berechnung von `Teilgerg * x`:“

```
if ((MAX_FLT/Teilerg) >= x) then
     Teilerg := Teilerg * x;
```

Andere Alternative: Trap-Handler, Exception-Handler (Prinzip: „Rechne und frage eventuell entstehende Fehler ab“) [5].

etwa in Eiffel:

```
local
     overflow: BOOLEAN is false
require
     -- Vorbedingung
do
     if not overflow then x := sqr(x)
     else ... end
rescue
     overflow := true;
     retry
ensure
     -- Nachbedingung
end
⋮
```

[5]Exception-Handler in der Fortran 90-Erweiterung und in ADA werden neben der Benutzung für „echte“ Ausnahmesituationen auch zur Implementierung von asynchronen — Ereignis- (event-) orientierten — Eingabeabläufen etwa bei Window-Schnittstellen von EDV-Applikationen genutzt.

oder in der von der IFIP-Arbeitsgruppe vorgeschlagenen Fortran 90-Erweiterung:

```
IO_CHECK:
     ENABLE (IO_ERROR, END_OF_FILE)
          ⋮
          READ (*, '(I5)') I
          READ (*,'(I5)', END = 90) J
          ⋮
       90 J = 0
     HANDLE (END_OF_FILE)
          WRITE (*, *) 'UNEXPECTED END-OF-FILE'
          STOP
     HANDLE (IO_ERROR)
          WRITE (8, *) 'I/O ERROR'
          STOP
     END ENABLE IO_CHECK
```

beziehungsweise:

```
     ENABLE
          MATRIX1 = FAST_INV (MATRIX)
     HANDLE DEFAULT
          ENABLE
               MATRIX1 = SLOW_INV (MATRIX)
          HANDLE DEFAULT
               WRITE (*, *) 'Cannot invert matrix'
               STOP
          END ENABLE
     END ENABLE
```

oder schließlich:

```
REAL FUNCTION CABS (Z)

     COMPLEX Z
     REAL S,ZI,ZR
     INTEGER I      ! Flags overflow iff equal -1
     INTRINSIC REAL, AIMAG, SQRT, ABS, MAX, CONDITION_SET

     I = 0

     ENABLE (OVERFLOW, UNDERFLOW)
          ZR = REAL(Z)
          ZI = AIMAG(Z)
          CABS = SQRT(ZR**2 + ZI**2)
     HANDLE (OVERFLOW, UNDERFLOW)

          ENABLE (OVERFLOW, UNDERFLOW)
               S = MAX(ABS(ZR), ABS(ZI))
               CABS = S*SQRT( (ZR/S)**2 + (ZI/S)**2 )
          HANDLE (OVERFLOW)
               I = -1
          HANDLE (UNDERFLOW)
               CABS = S
          END ENABLE

     END ENABLE

     IF (I == -1) CALL CONDITION_SET('OVERFLOW', I)

END FUNCTION CABS
```

1.3.5 Fallunterscheidungen und IEEE NaN's:

```
y1 = 0;
y2 = 0;
for (i=0;i<ITERATIONS;i++)
{
     x1 = y1;
     x2 = y2;

     y1 = x1 * x1 - x2 * x2 +c1;
     y2 = 2.0 * x1 * x2 + c2;

     n2 = y1 * y1 + y2 * y2;
     if (n2 > 4.0) then
          /* skip */

     else
          plot(y1,y2);

}
```

Einsetzen von

```
else if unordered (n2,4.0) then
     /* skip */
else
     plot(y1,y2);
```

beachtet „unordered“ Argumente! (Konvergente Iterationspunkte sollen geplottet werden. Ein Wert größer als 4.0 kann als Divergenzkriterium gewertet werden. Im Falle $\infty - \infty = NaN$ ist jedoch der Fall der Divergenz ohne Erfüllung von $n2 > 4.0$ gegeben.)

1.3.6 Portable und sichere numerische Software durch Programmiersprachen mit LIA-konformen Arithmetiken

(LIA = language independent arithmetic, draft international standard iso/iec cd 10967-1:1992)

A) Integer-Typen:

Ein Integer-Typ I ist eine Teilmenge von $\mathbb{Z}$, die durch folgende Parameter gekennzeichnet ist:

$$bounded \in \mathbb{B} \qquad \text{(ist } I \text{ endlich?)}$$

falls $bounded =$ **true**, dann zusätzlich

$$\begin{array}{ll} minint \in \mathbb{Z} & \text{(kleinste Zahl aus } I\text{)} \\ maxint \in \mathbb{Z} & \text{(größte Zahl aus } I\text{)} \end{array}$$

Dabei muß gelten: $maxint > 0$ und eine der drei Aussagen

$$\left\{ \begin{array}{ll} minint = 0 & \text{oder} \\ minint = -(maxint) & \text{oder} \\ minint = -(maxint + 1) & \end{array} \right.$$

sowie

$$I = \{x \in \mathbb{Z} \mid minint \leq x \leq maxint\}.$$

Falls $minint < 0$, so wird I *signed*, ansonsten *unsigned* genannt. Jeder Integer-Typ, für den *bounded* gleich **false** ist, muß *signed* sein. Jede dem LIA-Standard genügende Arithmetik muß mindestens einen *signed* Integer-Typ bereitstellen.

Folgende Operationen müssen für jeden Integer-Typ bereitgestellt werden:

$$\begin{array}{lll} add_I\colon & I \times I & \to I \cup \{\textbf{integer\_overflow}\} \\ sub_I\colon & I \times I & \to I \cup \{\textbf{integer\_overflow}\} \\ mul_I\colon & I \times I & \to I \cup \{\textbf{integer\_overflow}\} \\ div_I\colon & I \times I & \to I \cup \{\textbf{integer\_overflow, undefined}\} \\ rem_I\colon & I \times I & \to I \cup \{\textbf{undefined}\} \\ mod_I\colon & I \times I & \to I \cup \{\textbf{undefined}\} \\ neg_I\colon & I & \to I \cup \{\textbf{integer\_overflow}\} \\ abs_I\colon & I & \to I \cup \{\textbf{integer\_overflow}\} \\ sign_I\colon & I & \to I \\ eq_I\colon & I \times I & \to \mathbb{B} \\ neq_I\colon & I \times I & \to \mathbb{B} \\ lss_I\colon & I \times I & \to \mathbb{B} \\ leq_I\colon & I \times I & \to \mathbb{B} \\ gtr_I\colon & I \times I & \to \mathbb{B} \\ geq_I\colon & I \times I & \to \mathbb{B} \end{array}$$

Falls $minint = 0$, so darf neg_I, abs_I und $sign_I$ fehlen. Jeder Integer-Typ muß eine eigene solche Operationsschar haben.

Rundungen:

Eine **Rundung** rnd ist eine Abbildung der reellen Zahlen $\mathbb{R}$ auf eine Teilmenge X von reellen Zahlen mit:

- $u \in X \;\Rightarrow\; rnd(u) = u$
- $x_1 < u < x_2 \;\Rightarrow\; rnd(u) = x_1$ oder $rnd(u) = x_2$, falls $\nexists x \in X : x_1 < x < x_2$
- $u < z < rnd(u) \;\Rightarrow\; rnd(z) = rnd(u)$
- $rnd(u) < z < u \;\Rightarrow\; rnd(z) = rnd(u)$
- $x < y \;\Rightarrow\; rnd(x) \leq rnd(y)$

Beispiele:

a) $rnd(u) := \lfloor u \rfloor$ für $rnd : \mathbb{R} \to \mathbb{Z}$

b) $rnd(u) := \lfloor u + .5 \rfloor$ für $rnd : \mathbb{R} \to \mathbb{Z}$

Das Verhalten von div_I und rem_I ist durch die zugrundelegende Rundung

$$rnd_I : \mathbb{R} \to \mathbb{Z}$$

festgelegt. Jede konforme Arithmetik-Implementierung muß eine oder beide der Rundungen

$$rnd_I(x) \leq x \qquad \text{(Rundung nach } -\infty\text{)}$$
$$|rnd_I(x)| \leq |x| \qquad \text{(Rundung nach 0)}$$

bereitstellen. Jede Rundung erzeugt ein eigenes Paar von div_I und rem_I-Operationen.

Axiome: Für alle $x, y \in I$ muß gelten:

$$
\begin{array}{lll}
add_I(x,y) & = x + y & \text{falls } x + y \in I \\
 & = \textbf{integer\_overflow} & \text{falls } x + y \notin I \\
sub_I(x,y) & = x - y & \text{falls } x - y \in I \\
 & = \textbf{integer\_overflow} & \text{falls } x - y \notin I \\
mul_I(x,y) & = x * y & \text{falls } x * y \in I \\
 & = \textbf{integer\_overflow} & \text{falls } x * y \notin I \\
div_I(x,y) & = rnd(x/y) & \text{falls } y \neq 0 \text{ und } rnd_I(x/y) \in I \\
 & = \textbf{integer\_overflow} & \text{falls } y \neq 0 \text{ und } rnd_I(x/y) \notin I \\
 & = \textbf{undefined} & \text{falls } y = 0 \\
rem_I(x,y) & = x - (rnd(x/y) * y) & \text{falls } y \neq 0 \\
 & = \textbf{undefined} & \text{falls } y = 0
\end{array}
$$

Eine Implementierung muß eine oder beide von mod_I^1 und mod_I^2 bereitstellen:

$$
\begin{array}{lll}
mod_I^1(x,y) & = x - (\lfloor x/y \rfloor * y) & \text{falls } y \neq 0 \\
 & = \textbf{undefined} & \text{falls } y = 0 \\
mod_I^2(x,y) & = x - (\lfloor x/y \rfloor * y) & \text{falls } y > 0 \\
 & = \textbf{undefined} & \text{falls } y \leq 0 \\
neg_I(x) & = -x & \text{falls } -x \in I \\
 & = \textbf{integer\_overflow} & \text{falls } -x \notin I \\
abs_I(x) & = |x| & \text{falls } |x| \in I \\
 & = \textbf{integer\_overflow} & \text{falls } |x| \notin I \\
sign_I(x) & = 1 & \text{falls } x > 0 \\
 & = 0 & \text{falls } x = 0 \\
 & = -1 & \text{falls } x < 0 \\
eq_I(x,y) & = \textbf{true} & \Leftrightarrow \; x = y \\
neq_I(x,y) & = \textbf{true} & \Leftrightarrow \; x \neq y \\
lss_I(x,y) & = \textbf{true} & \Leftrightarrow \; x < y \\
leq_I(x,y) & = \textbf{true} & \Leftrightarrow \; x \leq y \\
gtr_I(x,y) & = \textbf{true} & \Leftrightarrow \; x > y \\
geq_I(x,y) & = \textbf{true} & \Leftrightarrow \; x \geq y
\end{array}
$$

Bemerkungen:

- $rem_I(x,y)$ kann nicht mittels

  $$sub_I(x, mul_I(div_I(x,y), y))$$

  berechnet werden, da div_I zum Overflow führen kann, rem_I jedoch nicht!

- Es gilt für alle rem_I-Versionen:

  $$
  \begin{array}{ll}
  x = div_I(x,y) * y + rem_I(x,y) & \forall y \neq 0 \\
  0 \leq |rem_I(x,y)| < |y| & \forall y \neq 0
  \end{array}
  $$

- mod_I erfüllt:

$$\begin{array}{ll} 0 \leq mod_I^j(x,y) < y & \forall y > 0\, \forall j \in \{1,2\} \\ y < mod_I^1(x,y) \leq 0 & \forall y < 0 \end{array}$$

B) Gleitkomma-Typen:

Ein Gleitkomma-Typ F ist eine endliche Teilmenge von $\mathbb{R}$, die durch die folgenden fünf Parameter gekennzeichnet ist:

$r \in \mathbb{Z}$	(Basis der Zahldarstellung)
$p \in \mathbb{Z}$	(Genauigkeit / Präzision von F, Anzahl der signifikanten Stellen)
$emin \in \mathbb{Z}$	(kleinster Exponent von F)
$emax \in \mathbb{Z}$	(größter Exponent von F)
$denorm \in \mathbb{B}$	(enthält F denormalisierte Werte?)

Dabei muß gelten:

$$\begin{array}{lll} r \geq 2 & \text{und} & p \geq 1, \\ emin \leq 2 - p & \text{und} & emax \geq 1, \end{array}$$

damit *epsilon* und 1.0 repräsentierbar sind,

$$\begin{array}{ll} F_N & \{0, \pm i * r^{e-p} \mid i, e \in \mathbb{Z}, r^{p-1} \leq i \leq r^p - 1, emin \leq e \leq emax\} \\ F_D & \{\pm i * r^{e-p} \mid i, e \in \mathbb{Z}, 1 \leq i \leq r^{p-1} - 1, e = emin\} \\ F & = F_N \cup F_D \quad \text{falls } denorm = \textbf{true} \\ & = F_N \quad\quad\;\; \text{falls } denorm = \textbf{false} \end{array}$$

Die Elemente von F_N heißen *normalisierte* Gleitkommazahlen, diejenigen von F_D *denormalisierte* Gleitkommazahlen.
Zusätzliche wünschenswerte Eigenschaften sind:

r sollte gerade sein, damit „Rundung zum Nächsten“ einfach wird;

$r^{p-1} \geq 10^6$, damit mindestens 7 signifikante Stellen existieren;

$emin - 1 \leq -k * (p-1)$ mit $k \geq 2$ und k möglichst groß, damit $epsilon^k$ in F liegt;

$emax > k * (p-1)$, damit $epsilon^{-k}$ in F liegt;

$-2 \leq (emin - 1) + emax \leq 2$, damit $\sqrt{fmin_N * fmax}$ zwischen $\frac{1}{r}$ und r liegt, womit dann für „fast alle“ x in F_N auch $\frac{1}{x}$ in F_N liegt.

Folgende abgeleitete Konstanten müssen dem Programmierer zur Verfügung stehen:

$$\begin{array}{lll} fmax & = max\{z \in F \mid z > 0\} & = (1 - r^{-p}) * r^{emax} \\ fmin_N & = min\{z \in F_N \mid z > 0\} & = r^{emin-1} \\ fmin_D & = min\{z \in F_D \mid z > 0\} & = r^{emin-p} \\ fmin & = min\{z \in F \mid z > 0\} & = fmin_D \text{ falls } denorm = \textbf{true} \\ & & = fmin_N \text{ falls } denorm = \textbf{false} \\ epsilon & = r^{1-p} & \text{(der maximale relative Fehler in } F_N) \end{array}$$

Mit

$$F^* = F_N \cup F_D \cup \{\pm i * r^{e-p} \mid i, e \in \mathbb{Z}, r^{p-1} \leq i \leq r^p - 1, e > emax\}$$

steht der Beschreibung (Axiomatik) der Operationen in F ein im Exponenten unbeschränkter Zahltyp zur Verfügung. Daneben sei zur metasprachlichen Beschreibung definiert:

$$\begin{array}{lll} e_F(x) & = \lfloor log_r|x| \rfloor + 1 & \text{falls } |x| \geq fmin_N \\ & = emin & \text{falls } |x| < fmin_N \end{array}$$

(Das ist der Exponent e in der Definition von F^*, F_N, F_D und per Definition gleich $emin$ für $x = 0$.)

Für den Gleitkomma-Typ müssen folgende Operationen bereitgestellt werden:

$$\begin{array}{lll}
add_F: & F \times F & \to F \cup \{\textbf{floating_overflow, underflow}\} \\
sub_F: & F \times F & \to F \cup \{\textbf{floating_overflow, underflow}\} \\
mul_F: & F \times F & \to F \cup \{\textbf{floating_overflow, underflow}\} \\
div_F: & F \times F & \to F \cup \{\textbf{floating_overflow, underflow, undefined}\} \\
neg_F: & F & \to F \\
abs_F: & F & \to F \\
sign_F: & F & \to F \\
exponent_F: & F & \to I \cup \{\textbf{undefined}\} \\
fraction_F: & F & \to F \\
scale: & F \times I & \to F \cup \{\textbf{floating_overflow, underflow}\} \\
succ_F: & F & \to F \cup \{\textbf{floating_overflow}\} \\
pred_F: & F & \to F \cup \{\textbf{floating_overflow}\} \\
ulp_F: & F & \to F \cup \{\textbf{underflow, undefined}\} \\
trunc_F: & F \times I & \to F \cup \{\textbf{undefined}\} \\
round_F: & F \times I & \to F \cup \{\textbf{floating_overflow, undefined}\} \\
intpart_F: & F & \to F \\
fractpart_F: & F & \to F \\
eq_F: & F \times F & \to \mathbb{B} \\
neq_F: & F \times F & \to \mathbb{B} \\
lss_F: & F \times F & \to \mathbb{B} \\
leq_F: & F \times F & \to \mathbb{B} \\
gtr_F: & F \times F & \to \mathbb{B} \\
geq_F: & F \times F & \to \mathbb{B}
\end{array}$$

Dabei muß I ein Integer-Typ sein, der mindestens die Werte $\pm(emax - emin + p - 1)$ enthält (für „*scale*“).

Rundungen (zur metasprachlichen Beschreibung):

Jede Rundung $rnd_F: \mathbb{R} \to F^*$ muß symmetrisch sein:

$$rnd_F(-x) = -rnd_F(x)$$

Für alle $x \in \mathbb{R}$ und $i \in \mathbb{Z}$ mit $|x| \geq fmin_N$ und $|x * r^i| \geq fmin_N$ gelte:

$$rnd_F(x * r^i) = rnd_F(x) * r^i$$

(d.h. die Rundung sei nur von der Mantisse abhängig).

Falls für $x \in \mathbb{R}$ und ein $i \in \mathbb{Z}$ mit $|x| < fmin_N$ sowie $|x * r^i| \geq fmin_N$

$$rnd_F(x * r^i) \neq rnd_F(x) * r^i$$

gilt, so spricht man von einem **Denormalisations-Signifikanzverlust** bei x.

rnd_error sei die kleinste Zahl aus F mit

$$|x - rnd_F(x)| \leq rnd\_error * r^{e_F(rnd_F(x))-p}$$

für alle $x \in \mathbb{R}$. Da rnd_F eine Rundung ist, gilt:

$$rnd\_error \leq 1$$

rnd_F rundet nach Null, falls für alle $x \in \mathbb{R}$ gilt:

$$|rnd_F(x)| \leq |x|.$$

rnd_F rundet zum nächsten Nachbarpunkt, falls für alle $x \in \mathbb{R}$ gilt:

$$|rnd_F(x) - x| \leq \tfrac{1}{2} * r^{e_F(x)-p}$$

Es muß eine abgeleitete Konstante *rnd_style* existieren mit einem der Werte:

near	, falls rnd_F zum nächsten Nachbarpunkt rundet;
truncate	, falls rnd_F nach Null rundet;
other	, sonst .

Falls $add_F^*(x, y)$ nicht mit $x + y$ für alle $x, y \in F$ übereinstimmt, so muß *rnd_style* = **other** und *rnd_error* = 1 sein.

Bemerkung: Übliche Rundungen sind (z.B. IEEE u.ä.):

a) Rundung nach $-\infty$

b) Rundung nach $+\infty$

c) Rundung nach Null

d) IEEE-Rundung zum nächsten Nachbarn:

Im Falle, daß ein Wert exakt zwischen zwei Nachbarn liegt, wird das „gerade“ Ergebnis bevorzugt; d.h. für $x \in F_N$, $u = r^{e_p(x)-p}$ gilt:

$$\begin{aligned} rnd(x + \tfrac{1}{2}u) &= x + u \quad \text{, falls } x/u \text{ ungerade,} \\ &= x \qquad \text{, falls } x/u \text{ gerade.} \end{aligned}$$

e) Traditionelle Rundung zum nächsten Narbarn:

$$rnd(x + \tfrac{1}{2}u) = x + u \qquad (\textit{weg von Null})^6.$$

a) und b) kommen wegen fehlender Symmetrie nicht für rnd_F in Frage!

Es gilt:

$$rnd\_error = \begin{cases} 1 & \text{für a), b) ,c)} \\ \frac{1}{2} & \text{für d) ,e)} \end{cases}$$

[6] gilt nur bei nichtnegativem x

Axiome: Für alle $x, y \in F$, $n \in I$ muß gelten:

$add_F(x,y)$	$= result_F(add_F^*(x,y), rnd_F)$	
$sub_F(x,y)$	$= add_F(x,-y)$	
$mul_F(x,y)$	$= result_F(x*y, rnd_F)$	
$div_F(x,y)$	$= result_F(x/y, rnd_F)$	falls $y \neq 0$
	$=$ **undefined**	falls $y = 0$
$neg_F(x)$	$= -x$	
$abs_F(x)$	$= \lvert x \rvert$	
$sign_F(x)$	$= 1$	falls $x > 0$
	$= 0$	falls $x = 0$
	$= -1$	falls $x < 0$
$exponent_F(x)$	$= \lfloor log_r \lvert x \rvert \rfloor + 1$	falls $x \neq 0$
	$=$ **undefined**	falls $x = 0$
$fraction_F(x)$	$= x / r^{exponent_F(x)}$	falls $x \neq 0$
	$= 0$	falls $x = 0$
$scale_F(x,n)$	$= result(x * r^n, rnd_F)$	
$succ_F(x)$	$= min\{z \in F \mid z > x\}$	falls $x \neq fmax$
	$=$ **floating_overflow**	falls $x = fmax$
$pred_F(x)$	$= max\{z \in F \mid z < x\}$	falls $x \neq -fmax$
	$=$ **floating_overflow**	falls $x = -fmax$
$ulp_F(x)$	$= r^{e_F(x)-p}$	falls $x \neq 0$ und $r^{e_F(x)-p} \in F$
	$=$ **underflow**	falls $x \neq 0$ und $r^{e_F(x)-p} \notin F$
	$=$ **undefined**	falls $x = 0$
$trunc_F(x,n)$	$= \lfloor x / r^{e_F(x)-n} \rfloor * r^{e_F(x)-n}$	falls $x \geq 0$
	$= -trunc_F(-x,n)$	falls $x < 0$

Für Beschreibungszwecke sei $rn_F\colon F \times \mathbb{Z} \to F^*$ definiert als

$$rn_F(x,n) = sign_F(x) * \lfloor \lvert x \rvert / r^{e_F(x)-n} + 1/2 \rfloor * r^{e_F(x)-n}$$

$round_F(x,n)$	$= rn_F(x,n)$	falls $\lvert rn_F(x,x) \rvert \leq fmax$
	$=$ **floating_overflow**	falls $\lvert rn_F(x,x) \rvert > fmax$
$intpart_F(x)$	$= sign(x) * \lfloor \lvert x \rvert \rfloor$	
$fractpart_F(x)$	$= x - intpart_F(x)$	
$eq_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x = y$
$neq_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x \neq y$
$lss_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x < y$
$leq_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x \leq y$
$gtr_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x > y$
$geq_F(x,y)$	$=$ **true**	$\Leftrightarrow \; x \geq y$

Bemerkungen:

- *result* ist eine metasprachlich genutzte Funktion, die zusätzlich zur Rundung auch die Projektion in einen endlichen Exponentenbereich vornimmt:

$$result_F\colon \mathbb{R} \times (\mathbb{R} \to F^*) \to F \cup \{\textbf{floating\_overflow, underflow}\}$$

  Dabei ist das erste Argument der zu „rundende“ Wert und das zweite die zu benutzende Rundungsfunktion:

$result_F(x, rnd)$	$= rnd(x)$	$\forall\ x = 0$ oder $fmin_N \leq \lvert x\rvert \leq fmax$,
	$= rnd(x)$	$\forall\ \lvert x\rvert > fmax$ und $\lvert rnd(x)\rvert = fmax$,
	$=$ **floating_overflow**	$\forall\ \lvert x\rvert > fmax$ und $\lvert rnd(x)\rvert \neq fmax$,
	$= rnd(x)$, **underflow**	$\forall\ 0 < \lvert x\rvert < fmin_N$ und $rnd(x) \in F$ und kein Denormalisationsverlust tritt auf (Wahl je nach Implementierung),
	$=$ **underflow**	$\forall 0 < \lvert x\rvert < fmin_N$ und Denormalisationsverlust tritt auf oder $rnd(x) \notin F$.

- Die oben metasprachlich genutzte approximative Addition

$$add_F^*\colon F \times F \to \mathbb{R}$$

  erlaubt auch einigen Hochleistungsimplementierungen (Vektorrechnen) noch LIA-konform zu sein. Diese liefern leider nicht immer das wünschenswerte Resultat

$$add_F(x, y) = result_F(x + y, rnd_F),$$

  sondern ein geringfügig anderes, da sie aus Performance-Gründen weder „sticky“ noch „rounding“ Bits benutzen. Arithmetiken, die auch noch das „guard digit“ nicht benutzen, können jedoch nicht LIA-konform sein, da add_F^* folgende Eigenschaften besitzen muß (damit $rnd_F(add_F^*(u, v))$ zumindest einen der benachbarten beiden Rasterpunkte von $u + v$ in F liefert):

  $\forall u, v, x, y \in F\ \forall i \in \mathbb{Z}$
  $add_F^*(u, v) = add_F^*(v, u)$
  $add_F^*(-u, -v) = -add_F^*(u, v)$
  $x \leq u + v \leq y \ \Rightarrow\ x \leq rnd_F(add_F^*(u, v)) \leq y$
  $u \leq v \ \Rightarrow\ add_F^*(u, x) \leq add_F^*(v, x)$
  Falls $u, v, u * r^i,$ und $v * r^i$ alle in F_N liegen, $add_F^*(u * r^i, v * r^i) = add_F^*(u, v) * r^i$

  add_F^* sollte am besten mit $add_F^*(x, y) = x + y$ übereinstimmen.

Zitat zum Zweck von add_F^*:

The typical approximate addition implementation described below shows the interaction between alignment, negation, and guard digits.

Floating point additions and subtractions form two cases, depending on the signs of the operands: implied additions (addition of operands of the same sign or subtraction of operands with different signs) and implied subtractions (addition with opposite signs, subtraction with like signs). In both implied addition and subtraction, the radix points of the operands are aligned by right-shifting the "smaller" operand's fraction. In an implied subtraction, the smaller operand must also be negated, but there is a performance cost associated with negating before the alignment-shift. To negate after alignment, enough guard digits must be maintained to propagate the borrow correctly even if digits are "lost" in the alignment, but this too has a cost in hardware and may also degrade performance. The minimum hardware needed is one guard digit for the implied addition case, and one guard digit and one "sticky bit" (which records whether informationwas lost during alignment) for the implied subtraction. To round-to-nearest, a "rounding bit" is also needed (to record the size of the next guard digit relative to $r/2$). This design would produce a sum that is accurate enough to produce a correct estimate of the perfect result and to round it correctly.

High-performance implementations frequently omit both the rounding bit and the sticky bit. This increases the execution speed and simplifies the hardware design, but the "smaller" operand may have lost some precision during the alignment and negation. The slight loss of percision can become visible to the programmer when the "optimized" intermediate result unexpectedly rounds to the "other" one of the two elements of F most closely bracketing the true result. In fact, this intermediate result itself may lie outside of the interval that contains the true result, but it is required to round to one of the endpoint, so that the magnitude of the error bound is not increased. Note that an implementation that omits the guard digit as well cannot guarantee that the rounded intermediate result will be one of the endpointas of the interval that contains the true sum, and therefore such an implementation cannot conform to the LIA-1.

- Falls eine Implementierung mehr als einen Gleitkomma-Typ gleicher Basis bereitstellt (seien etwa $F_1, F_2, F_3, \ldots$ solche mit den Eigenschaften $p_1 \leq p_2 \leq p_3 \ \ldots, \ emin_1 \geq emin_2 \geq emin_3 \ \ldots$ und $emax_1 \leq emax_2 \leq emax_3 \ \ldots$), so haben folgende Eigenschaften für numerische Zwecke besonders gute Auswirkungen:

  $2 * p_i \leq p_i + 1,$ (akurate Akkumulation von Residuen bei Iterationen, exakte Produkte für die Berechnung von Skalarprodukten und Normen)

  $2 * (emin_i - 1) \geq (emin_{i+1} - 1),$

  $2 * emax_i \leq emax_{i+1},$ (geringe Wahrscheinlichkeit eines Zwischenoverflows, obwohl das Endergebnis einer komplexen Rechnung evtl. noch in F liegt)

- Aus den Axiomen folgen die Eigenschaften:

  $\forall \Phi \in \{add_F, sub_F, mul_F, div_F, scale_F, cvt_{F' \to F}, cvt_{I \to F}\}$[7]

  $\forall u, v, x, y \in F, \forall j, h, n \in \mathbb{Z}, \varphi$ zu Φ gehörige exakte Operation:

  (I) $u \leq \varphi(x, y) \leq v \ \Rightarrow \ u \leq \Phi(x, y) \leq v$ oder Ausnahmebedingung.

  (II) $\varphi(x, y) \in F \ \Rightarrow \ \Phi(x, y) = \varphi(x, y)$ oder Ausnahmebedingung.

  (III) $\varphi(u, x) \leq \varphi(v, y) \ \Rightarrow \ \Phi(u, x) \leq \Phi(v, y)$ oder Ausnahmebedingung.

  (IV) $|\Phi(x, y) - \varphi(x, y)| \leq ulp(\varphi(x, y)) \leq ulp_F(\Phi(x, y))$ oder Ausnahmebedingung oder $\varphi(x, y) = 0$.

  (V) Im Falle $\Phi(x, y)$ in $\pm[fmin_N, fmax]$:
  $\left| \dfrac{\Phi(x, y) - \varphi(x, y)}{\varphi(x, y)} \right| \leq ulp_F(1) = epsilon$ oder Ausnahmebedingung.

  (VI) Im Falle $\varphi(x, y)$ und $\varphi(x * r^j, y * r^j)$ in $\pm[fmin_N, fmax] \cup \{0\}$:
  $\varphi(x * r^j, y * r^k) = \varphi(x, y) * r^n \ \Rightarrow \ \Phi(x * r^j, y * r^k) = \Phi(x, y) * r^n$ oder Ausnahmebedinung.

  Regel III kann im Falle einer approximierten Addition add_F^* von nicht Ideal-Form für add_F und sub_F verletzt sein. Dann gilt jedoch immer noch:

$$
\begin{aligned}
&u \leq v \ \Rightarrow \ add_F(u, x) \leq add_F(v, x)\\
&u \leq v \ \Rightarrow \ sub_F(u, x) \leq sub_F(v, x)\\
&u \leq v \ \Rightarrow \ sub_F(x, u) \geq sub_F(x, v)\\
&add_F(x, y) = add_F(y, x)\\
&mul_F(x, y) = mul_F(y, x)\\
&sub_F(x, y) = -sub_F(y, x)\\
&add_F(-x, -y) = -add_F(x, y)\\
&sub_F(-x, -y) = -sub_F(x, y)\\
&mul_F(-x, y) = mul_F(x, -y) = -mul_F(x, y)\\
&div_F(-x, y) = div_F(x, -y) = -div_F(x, y)
\end{aligned}
$$

[7] Alle anderen Operationen auf F sind exakt! Bei Operationen mit einem Argument interpretiere entsprechend (d.h. ignoriere „y“).

Für $x \neq 0$:

$$
\begin{array}{l}
x \in F_N \;\Rightarrow\; exponent_F(x) \in [emin, emax] \\
x \in F_D \;\Rightarrow\; exponent_F(x) \in [emin - p + 1, emin - 1] \\
r^{exponent_F(x)-1} \in F \\
r^{exponent_F(x)-1} \leq |x| < r^{exponent_F(x)} \\
fraction_F(x) \in [1/r, 1) \\
scale_F(fraction_F(x), exponent_F(x)) = x
\end{array}
$$

$scale_F(x, n)$ ist exakt ($= x * r^n$), falls $x * r^n$ in $\pm[fmin_N, fmax] \cup \{0\}$ liegt oder $n \geq 0$ und $|x * r^n| \leq fmax$ ist.

Für $x \neq 0, y \neq 0$:

$$
\begin{array}{l}
x = \pm i * ulp_F(x) \text{ für eine ganze Zahl } i \text{ mit:} \\
\qquad r^{p-1} \leq i < r^p \quad \text{if } x \in F_N \\
\qquad 1 \leq i < r^{p-1} \quad \text{if } x \in F_D \\
exponent_F(x) = exponent_F(y) \;\Rightarrow\; ulp_F(x) = ulp_F(y) \\
x \in F_N \;\Rightarrow\; ulp_F(x) = epsilon * r^{exponent_F(x)-1}
\end{array}
$$

Beachte, daß im Falle *denorm*=**true** die Funktion ulp_F für alle Gleitkommawerte ungleich Null definiert ist. Im anderen Falle führt ulp_F zum **underflow**, wenn das Argument kleiner als $fmin_N/epsilon$ ist, d.h. wenn $e_F(x) < emin + p - 1$.

Für $|x| \geq 1$ gilt:

$$intpart_F(x) = trunc_F(x, e_F(x)) = trunc(x, exponent_F(x))$$

Für jedes x gilt, falls keine Ausnahmebedingung eintritt:

$$
\begin{array}{lll}
succ_F(pred_F(x)) = x & & \\
pred_F(succ_F(x)) = x & & \\
succ_F(-x) = -pred_F(x) & & \\
pred_F(-x) = -succ_F(x) & & \\
succ_F(x) = x + ulp_F(x) & & \\
pred_F(x) \;\; = x - ulp_F(x) & \text{falls } x \neq r^n \text{ für ein } n \geq emin \\
\qquad\quad = x - ulp_F(x)/r & \text{falls } x = r^n \text{ für ein } n \geq emin \\
ulp_F(x) * r^{p-n} = r^{e_F(x)-n} &
\end{array}
$$

Ist zusätzlich $n > 0$, so folgt wenn keine Ausnahmebedingung auftritt:

$$
\begin{array}{lcl}
r^{exponent_F(x)-1} & \leq & |trunc_F(x, n)| \leq |x| \\
round_F(x, n) & = & trunc_F(x, n), \quad \text{oder} \\
 & = & trunc_F(x, n) + sign_F(x) * ulp_F(x) * r^{p-n}
\end{array}
$$

- Eigenschaften weiterer Operationen (z.B der transzendenten Funktionen) werden später in „LIA-Part 2“ festgelegt.

- *cvt* ist eine Konvertierungsfunktion mit den Eigenschaften:
  - Konversion zwischen zwei Integer-Typen hat werterhaltend zu wirken oder einen **overflow** zu erzeugen.
  - Konversion in einen Gleitkomma-Typ ist unter Zuhilfenahme der *result*-Funktion definiert, wobei eine Rundung „zum nächsten Nachbarn“ benutzt werden muß. Die Implementierung muß dokumentieren, welche solche Rundung sie benutzt.
  - Rundung in höhere Genauigkeit ist immer exakt.

Eine LIA-konform Arithmetik muß Konvertierungsfunktionen

a) zwischen zwei unterschiedlichen Integer-Typen,

b) zwischen zwei unterschiedlichen Gleitkomma-Typen derselben Basis,

c) zwischen Integer- und Gleitkomma-Typen

bereitstellen:

$\forall\ I_a \neq I_b$ Integer-Typen:

$$cvt_{I_a \to I_b} \colon I_a \to I_b \cup \{\textbf{integer\_overflow}\},$$

$$\begin{aligned} cvt_{I_a \to I_b}(x) &= x && \text{, falls } x \in I_b \\ &= \textbf{integer\_overflow} && \text{, sonst.} \end{aligned}$$

$\forall\ F_a \neq F_b$ Gleitkomma-Typen, $nearest_x \colon \mathbb{R} \to x$ Rundung „zum nächsten Nachbarn“:

$$\begin{aligned} &cvt_{F_a \to F_b} \colon F_a \to F_b \cup \{\textbf{floating\_overflow,underflow}\}, \\ &cvt_{F_a \to F_b}(x) = result_{F_b}(x, nearest_{F_b}) \end{aligned}$$

$\forall\ F$ Gleitkomma-Typ, I Integer-Typ:

$$cvt_{F \to I} \colon F \to I \cup \{\textbf{integer\_overflow}\};$$

$$\begin{aligned} cvt_{F \to I}(x) &= rnd_{F \to I}(x) && \text{, falls } rnd_{F \to I}(x) \in I \\ &= \textbf{integer\_overflow} && \text{, sonst} \end{aligned}$$

sowie:

$$\begin{aligned} &cvt_{I \to F} \colon I \to F \cup \{\textbf{floating\_overflow}\}, \\ &cvt_{I \to F}(x) = result_F(x, nearest_F). \end{aligned}$$

(Jede unterschiedliche Wahl von $rnd_{F\to I}$ und $nearest_F$ ergibt eine eigene Schar von Konvertierungsfunktionen.)

C) Ausnahmebedingungen (Notifikation):

In Sprachen mit Exception-Handlern müssen diese von LIA-konformen Arithmetiken unterstützt werden. Hier wird der Kontrollfluß geändert.
In Sprachen ohne Exception-Mechanismen muß (evtl. bei Programmende) ein „unübersehbares Prompt“ des Eintritts einer Ausnahmebedingung existieren. Daneben muß das Setzen der Ausnahmebedingungs-Flags

integer_overflow,

floating_overflow,

underflow,

undefined,

die bei Programmstart gelöscht werden und später nur durch explizite Programmaktion zurückgesetzt werden können, durch Unterprogramm-Aufruf abgefragt werden können:

$E = \{$**integer_overflow, floating_overflow, underflow, undefined,**$\ldots\}$,
$Ind =$ **set of** E,
clear_indicators: $Ind \to$
set_indicators: $Ind \to$
test_indicators: $Ind \to \mathbb{B}$
save_indicators: $\to Ind$

$\forall\ S \in Ind$ und dem aktuellen Flagstatus *IndStatus* gilt:

clear_indicators(S)	$IndStatus := IndStatus - S$
set_indicators(S)	$IndStatus := IndStatus \cup S$
test_indicators(S)	$(S \cap IndStatus \neq \{\}) \in \mathbb{B}$
save_indicators(S)	liefert *IndStatus* als Resultat

Bemerkungen:

- E darf mehr Ausnahmebedingungen enthalten, in IEEE_real etwa: **inexact**.
- Im Falle einer arithmetischen Ausnahmebedingung muß die Implementierung einen „Fortsetzungswert“ bereitstellen und die Operation des Programms fortsetzen:

Ausnahmebedingung	Fortsetzungswert
underflow	$rnd(x)$, falls *denrom* =**true** 0, sonst
integer_overflow **floating_overflow** **undefined**	Implementierungsabhängiger Wert, evtl. $\notin I \cup F$

Terminiert ein Programm bei noch gesetzten Ausnahmebedingungs-Flags, so muß diese Tatsache unter Nennung des Typs der Ausnahmebedingung und deren verursachender Operation vor Programmstop angezeigt werden!
Der Programmierer muß etwa durch Compiler-Flags die Wahl zwischen den soeben diskutierten Ausnahmebedingungs-Mechanismen haben.

D) <u>Dokumentation:</u>

Neben den verschiedenen Wahlmöglichkeiten etwa bzgl. *rnd*, muß eine LIA-konforme Arithmetik-Implementierung [8] folgendes dokumentieren:

- sprachabhängige Notation von Typen, Operationen, Notifikationen, Konstanten, Parametern,...
- Übersetzung(sreihenfolge) von arithmetischen Ausdrücken in eine Kombination der oben definierten Operationen.

1.3.7 LIA in C

(Neben LIA in C existieren bereits LIA-Bindings für Ada, Basic, CommonLISP, Fortran 77, Fortran 90, Modula-2, Pascal/Extended Pascal, PL/I.)

The programming language C is defined by ISO/IEC 9899:1990, *Information technology – Programming languages – C First Edition.*

An implementation should follow all the requirements of `LIA-1` unless otherwise specified by this language binding.

The operations or parameters marked "†" are not part of the language and must be provided by an implementation that wishes to conform to the `LIA-1`. For each of the marked items a suggested identifier is provided. An implementation that wishes to conform to the `LIA-1` must supply declarations of these items in a header file <`lia.h`>.

The `LIA-1` data type *Boolean* is implementated in the C `data type int` (1 = **true** and 0 = **false**).

Every implementation of C has signed integral types `int` and `long int` which conform to the LIA-1.

> NOTE – The conformity of `short int` and `signed char` is not relevant since values of these types are promoted to `int` before computations are done.

C has three floating point types that conform to this standard: `float, double,` and `long double`.

[8] vergleiche dazu Abschnitt 1.3.7

The parameters for an integer data type can be accessed by the following syntax:

maxint	INT_MAX	LONG_MAX
minint	INT_MIN	LONG_MIN

The parameter *bounded* is always **true**, and need not be provided.

The parameters for a floating point data type can be accessed by the following syntax:

r	FLT_RADIX			
p	FLT_MANT_DIG	DBL_MANT_DIG	LDBL_MANT_DIG	
emax	FLT_MAX_EXP	DBL_MAX_EXP	LDBL_MAX_EXP	
emin	FLT_MIN_EXP	DBL_MIN_EXP	LDBL_MIN_EXP	
denorm	FLT_DENORM	DBL_DENORM	LDBL_DENORM	†

The macro ∗_DENORM represents a boolean and must be defined to have values 1 or 0.

The C language standard presumes that all floating point precisions use the same radix and rounding style, so that only one indentifier for each is provided in the language.

The derived constants for the floating point data type can be accessed by the following syntax:

fmax	FLT_MAX	DBL_MAX	LDBL_MAX	
$fmin_N$	FLT_MIN	DBL_MIN	LDBL_MIN	
$fmin_D$	FLT_TRUE_MIN	DBL_TRUE_MIN	LDBL_TRUE_MIN	†
epsilon	FLT_EPSILON	DBL_EPSILON	LDBL_EPSILON	
rnd_error	FLT_RND_ERR	DBL_RND_ERR	LDBL_RND_ERR	†
rnd_style	FLT_ROUNDS			

The C standard specifies that the values of the parameters FLT_ROUNDS are from `int` with the following meaning in terms of the LIA-1 rounding styles.

nearest	FLT_ROUNDS > 0
truncate	FLT_ROUNDS $= 0$
other	FLT_ROUNDS < 0

NOTE – The definition of FLT_ROUNDS has been extended to cover the rounding style used in all LIA-1 operations, not just addition and subtraction.

All but one of the integer operations are either operators, or declared in the header file <stdlib.h>. The integer operations are listed below, along with the syntax used to invoke them:

add_I	$x + y$		
sub_I	$x - y$		
mul_I	$x * y$		
div_I	x / y		
rem_I	x % y		
mod_I^1	`modulo`(x, y)	`lmodulo`(x, y)	†
mod_I^2	`no binding`		
neg_I	$-\ x$		
abs_I	`abs`(x)	`labs`(x)	
$sign_I$	`sgn`(x)	`lsgn`(x)	†
eq_I	$x == y$		
neq_I	$x\ != \ y$		
lss_I	$x < y$		
leq_I	$x <= y$		
gtr_I	$x > y$		
geq_I	$x >= y$		

where x and y are expressions of type `int`.

The C standard permits div_I and rem_I (/ and %) to be implemented using either round toward zero or toward minus infinity. An implementation that wishes to conform to the LIA-1 must choose the same rounding for both and document the choice.

The floating point operations are either operators, or declared in the header file <`math.h`>. The operations are listed below, along with the syntax used to invoke them:

add_F	$x + y$			
sub_F	$x - y$			
mul_F	$x * y$			
div_F	x / y			
neg_F	$-\ x$			
abs_F	`fabsf`(x)†	`fabs`(x)	`fabsl`(x)†	
$sign_F$	`fsgnf`(x)	`fsgn`(x)	`fsgnl`(x)	†
$exponent_F$	`exponf`(x)	`expon`(x)	`exponl`(x)	†
$fraction_F$	`fractf`(x)	`fract`(x)	`fractl`(x)	†
$scale_F$	`scalef`(x, n)	`scale`(x, n)	`scalel`(x, n)	†
$succ_F$	`succf`(x)	`succ`(x)	`succl`(x)	†
$pred_F$	`predf`(x)	`pred`(x)	`predl`(x)	†
ulp_F	`ulpf`(x)	`ulp`(x)	`ulpl`(x)	†
$trunc_F$	`truncf`(x, n)	`trunc`(x, n)	`truncl`(x, n)	†
$round_F$	`roundf`(x, n)	`round`(x, n)	`roundl`(x, n)	†
$intpart_F$	`ffloor`(x)	`floor`(x)	`floorl`(x)†	
$fractpart_F$	`frcprtf`(x)	`frcprt`(x)	`frcprtl`(x)	
eq_F	$x == y$			
neq_F	$x\ != y$			
lss_F	$x < y$			
leq_F	$x <= y$			
gtr_F	$x > y$			
geq_F	$x >= y$			

where x and y are expressions of type `float, double` and `long double` and n is of type `int`.

> NOTE – $Scale_F$ can be computed using the `ldexp` libary function, only if `FLT_RADIX` = 2. The standard **C** function `frexp` differs from $exponent_F$ in that no notification is raised when the argument is 0.

An implementation that wishes to conform to the LIA-1 must provide the LIA-1 operations in all floating point precisions supported.

C provides the required type conversion operations with explicit cast operators:

$cvt_{F\to I}$, $cvt_{I'\to I}$	`(int)`x, `(long int)`x
$cvt_{I\to F}$, $cvt_{F'\to F}$	`(float)`x, `(double)`x, `(long double)`x

The C standard requires that float to integer conversions round toward zero. An implementation that wishes to conform to the LIA-1 must use round to nearest for converions to a floating point type.

An implementation that wishes to conform to the LIA-1 must provide recording of indicators as one method of notification. The data type *Ind* is identified with the data type `int`. The values representing individual indicators should be distinct non-negative powers of two and can be accessed by the following syntax:

integer_overflow	`INT_OVERFLOW`	†
floating_overflow	`FLT_OVERFLOW`	†
underflow	`UNDERFLOW`	†
undefined	`UNDEFINED`	†

The empty set can be denoted by 0. Other indicator subsets can be named by adding together individual indicators. For example, the indicator subset

{floating_overflow, underflow, integer_overflow}

would be denoted by the expression

```
FLT_OVERFLOW+UNDERFLOW+INT_OVERFLOW
```

The indicator interrogation and manipulation operations are listed below, along with the syntax used to invoke them:

set_indicators	`set_indicators`(i)	†
clear_indicators	`clear_indicators`(i)	†
test_indicators	`test_indicators`(i)	†
save_indicators	`save_indicators`()	†

where i is an expression of type `int` representing an indicator subset.

In addition, an implementation that wishes to conform to the LIA-1 shall provide the alternative of notification through termination with a message.

1.3.8 Zwischenrechnung in (unkontrollierter) höherer Genauigkeit?

IEEE_real erwähnt die Möglichkeit für implizite erhöhte Genauigkeit bei Zwischenrechnungen (etwa der Auswertung von arithmetischen Ausdrücken). Zur Zeit existieren hauptsächlich drei Quellen für solche implizite „Mehrfachgenauigkeit“:

i) Die ALU arbeitet mit 80 Bit-Registern (Intel 8087). Falls für eine Zwischenrechnung die Anzahl der ALU-Register ausreicht, so wird voll in diesem Format gerechnet. Anderenfalls weiß der Programmierer nicht, welche Teile der Rechnung im 80 Bit-Format und welche im 64 Bit- Format ausgeführt werden. Werde etwa $a \cdot b + c \cdot d$ berechnet, so kann dies geschehen durch:

$$\begin{aligned} T_1 &:= rnd_{64}(rnd_{80}(a \cdot b) + rnd_{80}(c \cdot d)), \\ T_2 &:= rnd_{64}(rnd_{80}(a \cdot b) + rnd_{64}(c \cdot d)), \\ T_3 &:= rnd_{64}(rnd_{64}(a \cdot b) + rnd_{80}(c \cdot d)), \\ T_4 &:= rnd_{64}(rnd_{64}(a \cdot b) + rnd_{64}(c \cdot d)). \end{aligned}$$

ii) Doppelt genaue Triadenberechnung der Form $p \cdot q + r$ etwa beim IBM RISC System/6000. Auch hier kann $a \cdot b + c \cdot d$ verschieden berechnet werden:

$$\begin{aligned} T_5 &:= rnd_{64}(rnd_{64}(a \cdot b) + c \cdot d), \\ T_6 &:= rnd_{64}(a \cdot b + rnd_{64}(c \cdot d)), \\ T_4 &:= rnd_{64}(rnd_{64}(a \cdot b) + rnd_{64}(c \cdot d)). \end{aligned}$$

iii) Doppelt genaue Akkumulation von Produkten (etwa bei der IBM 7094):

$$T_7 := rnd_{64}(a \cdot b + c \cdot d)$$

(vgl. auch Pascal XSC).

Vorteile:

- „exakteres“ Ergebnis als bei konsistenter konstanter Genauigkeit bei Zwischenrechnungen
- der Programmierer braucht explizit keine mehrfach genaue Arithmetik einsetzen

Nachteile:

- Programme sind weniger portabel, da die erweiterte Genauigkeit von Implementierung zu Implementierung variiert
- unvorhersagbare Resultate, da je nach Belegungsgrad der ALU- Register, der Compileroptimierung, etc. andere „Rundungsschritte“ durchgeführt werden
- Unmöglichkeit, Algorithmen in Abhängigkeit der Gleitkomma-Typen-Parameter zu konzipieren

- zu wenig Kontrolle, da die implizit berechneten Teilergebnisse einer höheren Genauigkeit nicht in temporäre Variablen gerettet werden können
- zusätzliche Genauigkeit kann nicht nach Wunsch zielgerecht angefordert werden
- Genauigkeit versus Geschwindigkeit kann nicht absichtlich eingesetzt werden

Deshalb fordert LIA, daß:

i) jeder Gleitkomma-Typ — auch der einer impliziten Zwischenrechnung — dokumentiert werden muß,

ii) die Umsetzung von Ausdrücken in eine Reihenfolge von Operationen (verschiedenster Genauigkeit) dokumentiert werden muß.

Implizite erhöhte Genauigkeit ist nicht immer wünschenswert, da sie Gesetzmäßigkeiten wie etwa das Kommutativgesetz wieder ungültig machen kann.

Ein Beispiel:

Sei $A := 12.34$, $B := 89.08$, $C := 23.45$, $D := -45.58$. Dann ist

$$A \cdot B + C \cdot D = 30.3962 \approx 30.40.$$

Bei Benutzung einer 4-stelligen Dezimalarithmetik gilt mit

$R_m := \ rnd_m(A \cdot B),\ S_n \ = \ rnd_n(C \cdot D),\ T_{mn} := \ rnd_4(R_m + S_n);$

0) LIA:

	$S_4 = -1069.0$
$R_4 = 1099.0$	$T_{44} = 30.00$

i) 5-stellige Register:

Environment 2	$S_4 = -1069.0$	$S_5 = -1068.9$
$R_4 = 1099.0$	$T_{44} = 30.00$	$T_{45} = 30.10$
$R_5 = 1099.2$	$T_{54} = 30.20$	$T_{55} = 30.30$

ii) Triaden:

Environment 3	$S_4 = -1069.0$	$S_8 = -1068.8510$
$R_4 = 1099.0$	$T_{44} = 30.00$	$T_{48} = 30.15$
$R_8 = 1099.2472$	$T_{84} = 30.25$	

iii) 8-stellige Akkumulation:

Environment 4	$S_4 = -1069.0$	$S_8 = -1068.8510$
$R_4 = 1099.0$	$T_{44} = 30.00$	$T_{48} = 30.15$
$R_8 = 1099.2472$	$T_{84} = 30.25$	$T_{88} = 30.40$

Konsequenz: IEEE_real bei vielen Implementierungen ist nicht einmal kommutativ bezüglich der Addition!

1.3.9 Konsequenzen von unbeachteten Arithmetik-Eigenschaften

Fatal Error: How Patriot Overlooked a Scud

Zitat ,,Science, 13.März 1992":

Even a minute mathematical error can lead to tragedy in the computer age, as confirmed by a report on the Patriot missile issued by the General Accounting Office (GAO) last week. The report describes how a minor bug in Patriot's software allowed an Iraqi Scud missile to slip through Patriot defences a year ago and hit U.S.Army in Dhahran, Saudi Arabia, killing 28 servicemen.

GAO undertook the study on orders from Representative Howard Wolpe (D-MI), who says he has questions whether the military's "logistical apparatus is adequate to support ... software-driven weapons." He was not reassured. "The episode," Wolpe wrote in a letter to Defence Secratary Richard Cheney, "makes clear the problems American troops may face as we continue to take advantage of the benefits of the computer revolution in developing weapons."

According to the GAO report, the patriot's electronic brain – now 20 years old – would have performed well in the task it was designed to do, which was to track and shoot down relatively slow-moving aircraft. But it ran into trouble when it was pressed into service in the Persian Gulf to defend against high-speed ballistic missiles. The main flaw was in the way the Patriot battery's missiletracking computers processed timing information, which affected its ability to pinpoint the location of fast-moving targets.

The computer's tracking calculations depended on signals from its internal clock, which it translated into a "floating point" mathematical value. Because the computer could handle only relatively small chunks of data (by today's standards), it was forced to truncate this time value slightly, creating a slight error. By itself, the flaw would not have been fatal, but the Patriot software was written in a way that caused the error to increase steadily as time passed on the computer's clock.

That's what happened on the night of 25 Feburary 1991. A Scud missle launched from Iraq popped over the horizon in Saudi Arabia and was picked up by a Patriot's radar, which was then performing a wide search of the sky. The Patriot locked onto this target and calculated a "track" that was an approximaton of the the path it would follow to the ground. To confirm that this was truly an enemy Scud, the computer was programmed to get a second radar sighting to determinate whether the object was following the path expected of a ballistic missile. If it was not, the signal would be rejected as a false alarm. And to speed up the process, the software told the computer to analyse only data from a small portion of the radar beam – the portion within a mathematically limited zone (the "range gate") centered on the path that a ballistic missile would be expected to follow. If the computer found a target within this range gate, it would know that the attack was real and would launch a Patriot missile. Sadly, in this case the computer miscalculated the position of the range gate, failed to see the Scud, and ruled that the original signal was a false alarm.

The mistake occured because this particular Patriot battery had been running continuously for about 100 hours. According to GAO, its logic had build up a timing lag of 0.3433 seconds. That may sound trivial, but when tracking targets traveling at ballistic speeds the error was fatal, for it caused the computer to shift the range gate 687 meters, letting the Scud pass unnoticed.

Ironically, about a week before the Dhahran tragedy, U.S. military officials had been warned that something like this could happen, according to GAO. The warning came first from the Israeli military, which had been analysing data records from Patriots batteries in Israel. The Israelis discovered that after about 8 hours of continuous use, the Patriot system built up a timing error of 0.0275 second, enough to create a range-finding error of about 55 meters. They passed the word to the U.S. Patriot project office on 11 February 1991.

Within a few days, the Patriot project office made a software fix correcting the timing error, and sent it out to the troops on 16 February 1991. On 21 February the office sent out a warning that "very long run times" could affect the targeting accuracy and alerted officers to the fact that new software was on the way. The troops were not told, however, how many hours "very long" was, or that it would help to switch the computer off and on again after 8 hours. The U.S. forces finally solved the timing problem when they received and installed the new software at Daharan on 26 February – a day too late. □ ELIOT MARSHALL

SIAM News July 1992

Roundoff Error and the Patriot Missile

By Robert Skeel

The March 13 issue of *Science* carried an article claiming, on the basis of a report from the General Accounting Office (GAO), that a "minute mathematical error ... allowed an Iraqi Scup missile to slip through Patriot missile defences a year ago and hit U.S. Army barracks in Dhahran, Saudi Arabia, killing 28 servicemen." The article continues with a readable account of what happened.

The article says that the computer doing the tracking calculations had an internal clock whose values were slightly truncated when converted to floating-point arithmetic. The error were proportional to the time on the clock: 0.0275 seconds after eight hours and 0.3433 seconds after 100 hours. A calculation shows each of these relative errors to be both very nearly 2^{-20}, which is approximately 0.0001%.

The GAO report contains some additional information. The internal clock kept time as an integer value in units of tenths of a second, and the computer's registers were only 24 bits long. This and the consistency in the time lags suggested that the error was caused by a fixed-point 24-bit representation of 0.1 in base 2. The base 2 representation of 0.1 is nonterminating: for the first 23 binary digits after the binary point, the value is $0.1\times(1-2^{-20})$. The use of $0.1\times(1-2^{-20})$ in obtaining a floating-point value of time in seconds would cause all times to be reduced by 0.0001%.

This does not really explain the tracking errors, however, because the tracking of a missile should depend not on the absolute clock-time but rather an the time that elapsed between two different radar pulses. And because of the consistency of the errors, this time difference should be in error by only 0.0001%, a truly insignificant amount.

Further inquiries cleared up the mystery. It turns out that the hypothesis concerning the truncated binary representation of 0.1 was essentially correct. A 24-bit representation of 0.1 was used to multiply the clock-time, yielding a result in a pair of 24-bit registers. This was transformed into a 48-bit floating-point number. The software used had been written in assembly language 20 years ago. When Patriot systems were brought into the Gulf conflict, the software was modified (several times) to cope with the high speed of ballistic missiles, for which the system was not originally designed.

At least one of these software modifications was the introduction of a subroutine for converting clock-time more accurately into floating-point. This calculation was needed in about half a dozen places in the program, but the call to the subroutine was not inserted at every point where it was needed. Hence, with a less accurate truncated time of one radar pulse being subtracted from a more accurate time of another radar pulse, the error no longer cancelled.

In the case of the Dhahran Scud, the clock had run up a time of 100 hours, so the calculated elapsed time was too long by $2^{-20}\times 100$ hours = 0.3433 seconds, during which time a Scud would be expected to travel more then half a kilometer.

The roundoff error, of course, is not the only problem that has been identified: serious doubts have been expressed about the ability of Patriot missiles to hit Scuds.

Robert Skeet is a professor of computer science at the University of Illinois at Urbana-Champaign.

1.4 Spezifikation für schnelle Prototypen

Graphische und/oder sprachliche Spezifikation von Algorithmen/Systemen:

heute: Funktionen und Schnittstellen der einzelnen Module von Systemen beschreiben (erwünschte/geforderte Eigenschaften)

→ präzise Formulierung

→ „Fallstudien“, (schnell entstandene) erste lauffähige Programme

→ Verifikation und Evaluation[9] vergleichsweise einfach

→ standardmäßig angewandt bei (graphischen) Benutzerschnittstellen, Datenmodellen (Datenbanken), „KI“

zukünftig: Interpretation, Simulierung oder Übersetzung in (imperative) Programmiersprachen.

1.4.1 Widget Creation Libary (X11)

Planung:

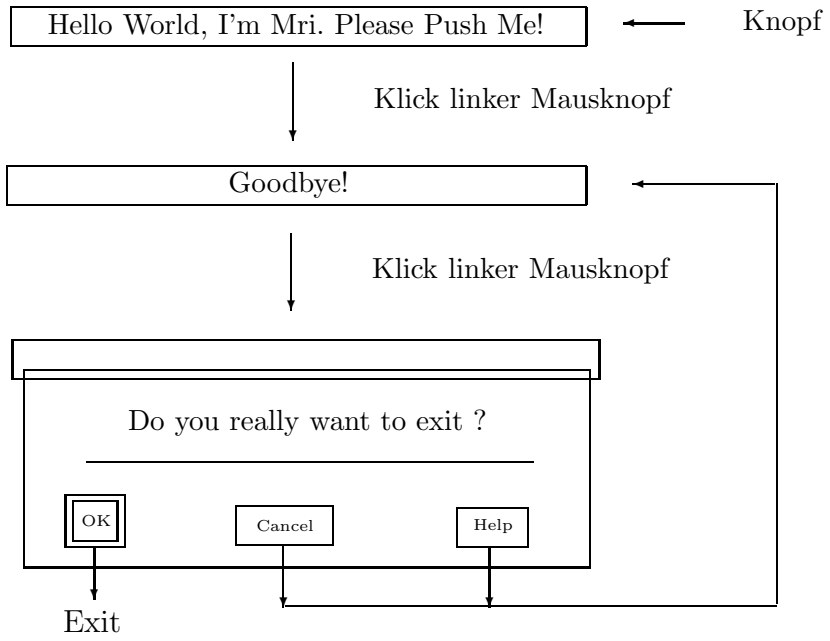


Abbildung 1.1: Benutzerschnittstelle (Plan)

[9] vergleiche Kapitel 2

abstrakte Beschreibung:

Mri.wcChildren:	exitDialog, push
Mri.title:	Mri using M2.Goodbye
∗push.wcClass:	xmPushButtonWidgetClass
∗push.labelString:	Hello World, I'm Mri. Please Push Me!
∗push.activateCallback:	WcSetValueCB(\ ∗push.activateCallbak: WcMenageCB(∗exitDialog.exitDialog) \ ); \ WcSetValueCB(∗push.labelString: Goodbye!)
∗exitDialog.wcConstructor:	XmCreateQuestionDialog
∗exitDialog.wcManaged:	False
∗exitDialog.messageString:	Do you really want to exit?
∗exitDialog.okCallback:	WcExitCB(1)

Interpretation oder Compilation der abstrakten Beschreibung:

Interpretation:	**Compilation:**
rapid Prototype	Effektivität
spätere Modifikationsmöglichkeit (Anwender-Konfigurierbarkeit)	weniger Speicherbedarf
	Gesamtbeschreibung syntaktisch vollständig geprüft

1.4.2 Interaktive graphikgestützte Generierung von Benutzerschnittstellen (X11):

Konstruktion und Test:

a) Beschreibung/Prototyp

b) Code

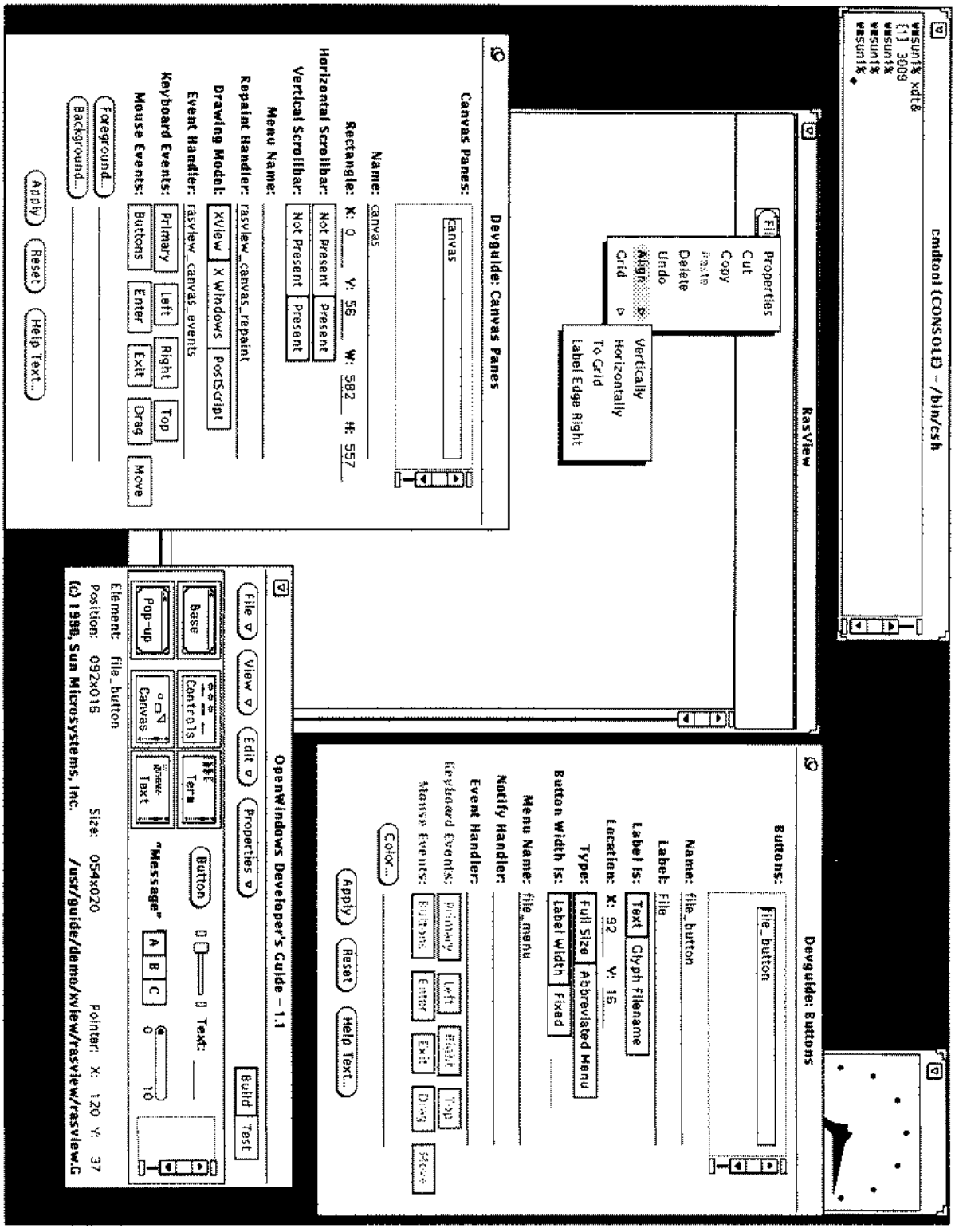
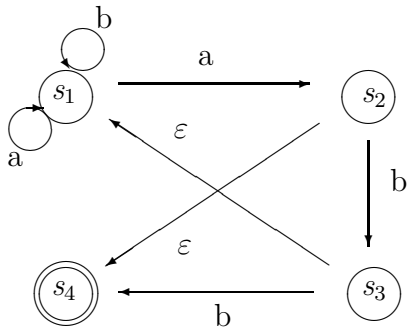


Abbildung 1.2: Design der Benutzerschnittstelle durch „drag and drop“

Abbildung 1.3: Test der rudimentären Aktionen der Benutzerschnittstelle

1.4.3 Prolog und prädikative Spezifikation

Der Programmierer muß die Fakten und Regeln benennen und die Lösung des anstehenden Problems so formulieren, daß es als Wahrheitsaussage vorliegt, die dann mit Hilfe der Fakten und Regeln entschieden werden kann.
Ein Beispiel zeigt, wie man einen endlichen Automaten prädikativ beschreiben kann. Dieser sei durch folgende Graphen gegeben:



Rest $= a$ oder $= abb$
Vorher: Beliebige Folgen von a's und b's

Abbildung 1.4: Transitionsdiagramm

Daß s_4 Endzustand ist und Übergänge von s_1 mittels a nach s_2, von s_1 mittels a nach s_1 etc. erfolgen, läßt sich durch Fakten ausdrücken. Auch die ε-Transitionen lassen sich durch Fakten darstellen:

Beschreibung einer Eingabesprache:

$$(a \mid b)^*(a \mid abb \mid ab(a \mid b)^* \ldots a \mid \ldots)$$

Fakten:

Sei $\kappa = \textit{Menge der Knoten} = \{s_1, s_2, s_3, s_4\}$ und $\vartheta = \textit{Menge der Buchstaben} = \{a, b\}$:

final (s_4). } einstellige Relation auf κ (Daten)

trans (s_1, a, s_2).
trans (s_1, a, s_1).
trans (s_1, b, s_1).
trans (s_2, b, s_3).
trans (s_3, b, s_4). } dreistellige Relation auf $\kappa \times \vartheta \times \kappa$

silent (s_2, s_4).
silent (s_3, s_1). } zweistellige Relation auf $\kappa \times \kappa$

Die Regeln beschreiben nun, ob man bei der Eingabe von Zeichenketten aus den Buchstaben a und b in den Endzustand s_4 kommt oder nicht.

Regeln:

$$\left.\begin{array}{ll} \text{accept } (S, & [\ \ ]) \text{ :-} \\ & \text{final } (S). \\ \text{accept } (S, & [X, \text{Rest}]) \text{ :-} \\ & \text{trans}(S, X, S_1), \\ & \text{accept } (S_1, \text{Rest}). \\ \text{accept } (S, & \text{String}) \text{ :-} \\ & \text{silent } (S, S_1), \\ & \text{accept } (S_1,\ \text{String}). \end{array}\right\} \text{Relation auf } \kappa \times \vartheta^*$$

Es sind zwei untereinanderstehende Zeilen der Regeln logisch durch „und wenn“ miteinander verbunden.

Abfragen:	? – accept	$(s_1, [a, b, b])$.	$\rightarrow$ wahr
	? – accept	$(s_1, [X_1, X_2, X_3])$.	$\rightarrow$ wahr für $X_1X_2X_3 = aaa, abb, \ldots$
	? – accept	$(X, [a, a, b, b])$.	$\rightarrow$ wahr für $X = s_1$ und s_3

Wie die beiden letzten Beispiele zeigen, dürfen dabei Variablen benutzt werden.

1.4.4 ISETL und Quantoren

```
[[i,i**2]:i in [100..104]];
[i: i in [1..200] and exists j in [1..50]|j*4=i];
N:=100;
[i: i in [2..N]| forall j in [2..i-1]| i mod j /= 0];
[%+ [1,3..2*i+1] : i in [1..10]];
```

steht in ISETL für die mathematischen Formulierungen:

$$\{[i, i^2\ ] \mid i \in \{100, \ldots, 104\}\}$$

$$\{i \in \{1, \ldots, 200\} \mid \exists\ j \in \{1, \ldots, 50\} :\ j * 4 = i\}$$

$$\{i \in \{2, \ldots, N\} \mid \forall\ j \in \{2, \ldots, i-1\} :\ i \bmod j \neq 0\}$$

$$\left(\sum_{j=0}^{i}(2j+1)\right)_{i=1}^{10}$$

1.4.5 SADT: „structured analysis and design technique“

- Jedes System wird sowohl durch seinen funktionellen

  $$\text{Eingabeobjekt} \longrightarrow \boxed{\text{verarbeite}} \longrightarrow \text{Ausgabeobjekt}$$

  als auch durch seinen objektorientierten Aspekt

  $$\text{erzeuge} \longrightarrow \boxed{\text{Objekt}} \longrightarrow \text{verbrauche}$$

  dargestellt.

  (Das erleichtert die Überprüfung des *graphischen Modells* auf Korrektheit und Vollständigkeit wesentlich!)

- Top–Down–Verfeinerung des Systems: (je Schritt ≤ 6 Teilsysteme)

- Jedes System/Teilsystem wird von mehreren verschiedenen Standpunkten aus betrachtet

- Da in SADT keine Kontrollflüsse beschrieben werden können, wird die abstrakte statt der algorithmischen Spezifikation gefördert!

Beispiel eines Verfeinerungsschritts unter funktionalem Aspekt:

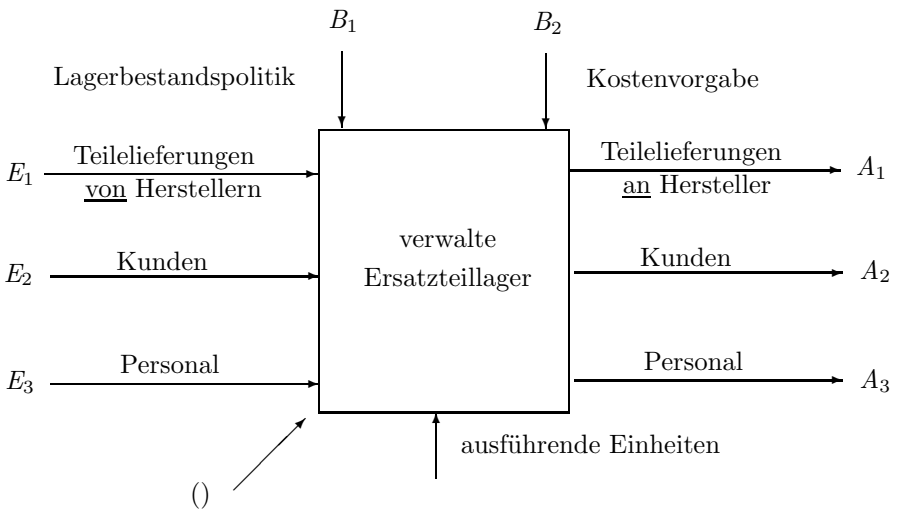


Abbildung 1.5: Objekte des übergeordneten Diagramms

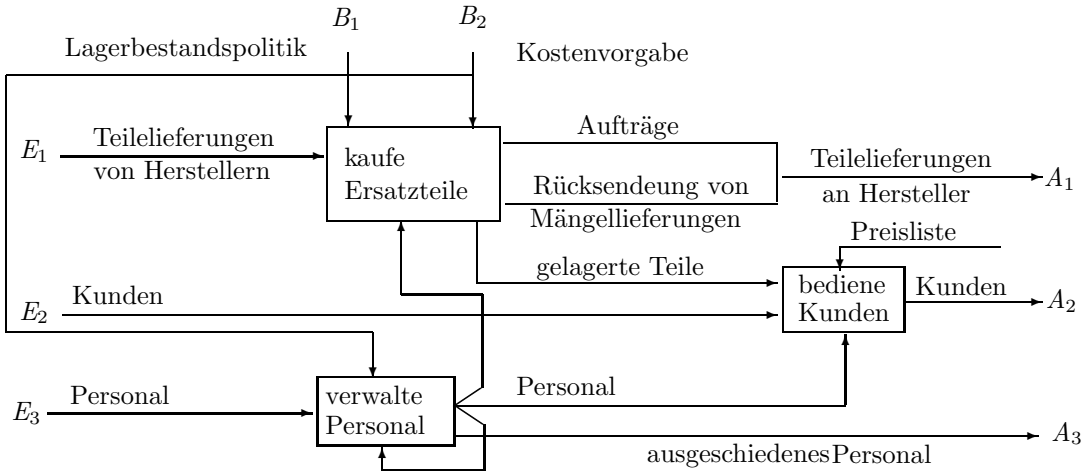


Abbildung 1.6: verfeinertes Diagramm

1.5 Formale Spezifikation von freien Eingabesprachen

(Eingabe, Ausgabe und ihr funktioneller Bezug)

1.5.1 Statusdiagramme und lexikalische Spezifikation

Eingabesprache / Grundobjekte einer Eingabesprache:

Der lexikalischer Aufbau kann beschrieben werden durch Statusdiagramme (Transitionsdiagramme)

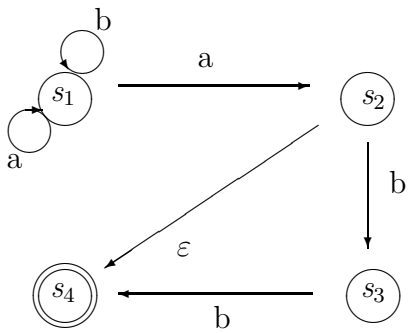


Abbildung 1.7: lexikalische Definition

oder

1.5.2 EBNF/Sytaxdiagramme und syntaktische Spezifikation

durch Benutzung der „extended Backus-Naur-Form“

Sytax einer Eingabesprache:

unsigned_real	=	unsigned_integer ”.” fractional_part
		[”e” scale_factor] |
		unsigned_integer ”e” scale_factor .
unsigned_integer	=	digit_sequence .
digit_sequence	=	”digit” {”digit”} .
		⋮

eventuell in Form von Syntaxdiagrammen

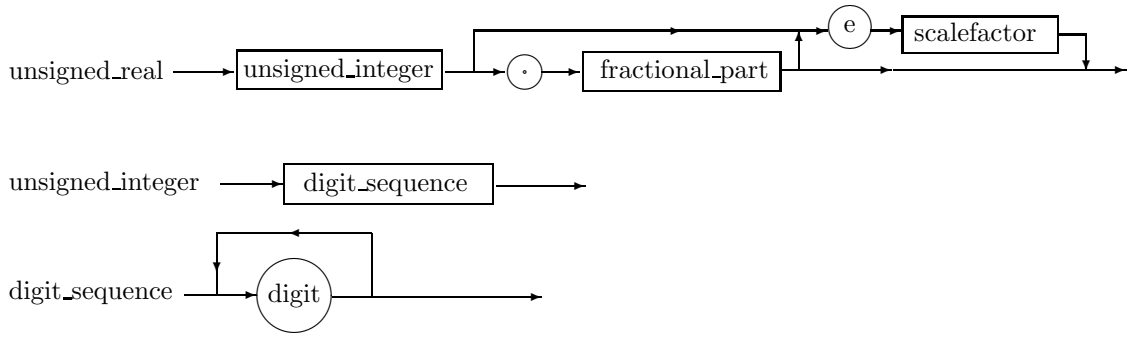


Abbildung 1.8: syntaktische Definition

beziehungsweise

1.5.3 Pseudocode zur Semantikbeschreibung

durch Pseudocode, wobei dieser auch noch zur Definition der **Semantik** (des funktionellen Bezugs) in einer sogar als Kommentar im Quellcode realisierbaren Form dienen kann:

Die Syntaxdefinition in EBNF

```
term = factor { multiplying_operator factor  } .
```

genauer

```
term = factor1 {   *      factor2   |
                   /      factor2   |
                   div    factor2   |
                   mod    factor2   |
                   and    factor2     }.
```

wird zur Semantikdefinition in Pseudocode:

```
if (type (factor_1) = integer) and (type (factor_2) = integer) then begin
     case multiplying_operator of
          div: type(term) ⇐ integer;
               value(term) ⇐ if value(factor_2) ≠ 0 then
                               value(term)=value(q) mit
                                        value(factor_1)=q*value(factor_2)+r und
                                        q,r ∈ ℤ, 0 ≤ r < |value(factor_2)|
                             else
                               value(term)=exception('Division by Zero')
          mod: ...
          * : ...      ⟵ overflow beachten!!
     end case;
end else if (type(factor_1) = integer) and (type(factor_2) = real) then begin
     ⋮
end
⋮
```

1.6 Spezifikation in gängigen Programmiersprachen

1.6.1 assert in C, C++

```
# include <assert.h>
     ⋮
int fakultaet(int i)
{
     assert (i>=0);     /* Vorbedingung */
          ⋮
     assert(result>0);     /* Nachbedingung */
}
```

„Assertion failed: file ass.c, line 15“ bei fakultaet(17) statt 17! zum Beispiel das Ergebnis -288522240.

Nach dem Austesten: eventuell Abstellen der Überprüfungen durch

```
# define NDEBUG
# include <assert.h>
     ⋮
```

und erneute Compilation.

1.6.2 Assertions in Eiffel

a) assertions (Zusicherungen, Prädikate):

`Positive: n>0; Not_void: not x.Void;` → benutzbar in b) ... e)

`Positive` → Label, das gegebenenfalls angezeigt wird vom Laufzeitsystem

„;“ steht für „and“

b) Vor- und Nachbedigungen von Prozeduren:

routine_name(argument declarations) `is`
`-- Header comment`
`require`
precondition
`do`
routine body, i.e. instructions
`ensure`
postcondition
`end --` *routine_name*

Beispiel:

```
insert(element: T, key: STRING) is
        Insert elements with key key
    require
        nb_elements < max_elements
    do
        ..."Insertion algorithm"...
    ensure
        nb_elements ≤ max_elements;
        value(key) = element;
        nb_elements = old nb_elements + 1
    end -- insert
```

`old` ist in der Nachbedigung erlaubt und bezeichnet den Wert seines Arguments bei Unterprogrammbeginn.

Fehlende Semantik, die in Eiffel nur in Form von Kommentaren möglich ist:

$\forall k \in$ old(keyvalues) : value(k) = old(value(k))

(Quantoren fehlen, d.h. eine volle Spezifikation, die ein Prädikatenkalkül 1.Stufe erfordert, ist leider unmöglich.)

Beispiel 2:

Class *STACK*

Ancestors of class *STACK:*

No ancestors.

-- Stacks (last-in, first-out),

-- without commitment to a particular representation

`deferred class interface` *STACK[T]* `exported features`

nb_elements, empty, full, top, push, pop, change_top, wipe_out

`feature specification`

---

nb_elements: INTEGER
-- Number of elements inserted
`deferred`

---

empty: BOOLEAN
-- Is stack empty?
`ensure`
Result = (nb_elements = 0) ⟵ volle Semantik

---

full: BOOLEAN
`-- Is stack full?`
`deferred`

---

top: *T*
`-- Last element pushed` ⟵ nur informell
`require`
not_empty: `not` *empty*
`deferred`

---

push(v:T)
`-- Push v onto stack.`
`require`
not_full: `not` *full*
`deferred`
`ensure`
`not` *empty*;
top = *v*;
nb_elements = `old` *nb_elements + 1* ⟵ „old values“ nur informell

---

pop
`-- Remove top element.`
`require`
not_empty: `not` *empty*
`deferred`
`ensure`
`not` *full*;
nb_elements = `old` *nb_elements − 1* ⟵ Änderung der Values des Stacks nur informell

---

change_top (v:T)
`-- Replace value of top element by v`
`require`
not_empty: `not` *empty*
`ensure`
`not` *empty*;
top = *v*
nb_elements = `old` *nb_elements*

wipe_out
  -- Remove all elements.
  deferred
  ensure
    empty

---

end interface -- class *STACK.*

Semantik für pop-Operationen auch auf einem leerem Stack:

⋮
ensure
  ((not old *empty)* **and** *(nb_elements =* **old** *nb_elements − 1)* **and not** *full)*
  or
  *((***old** *empty)* **and** *Nochange)* ⟵ Nochange in Nachbedingung erlaubt!

c) Gemeinsame Eigenschaften aller Objekte eines abstrakten Datentyps:

class *STACK[T]* **export**
feature ...
invariant
  0 <= nb_elements; nb_elements <= max_size;
  empty = (nb_elements =0)
end -- class *STACK*

Korrektheit der Klasse C $\overset{Def.}{\Longleftrightarrow}$

$$\begin{cases} \text{1.)} & \forall \text{ exportierte Routinen } r \neq \mathit{Create} \text{ und} \\ & \forall \text{ Sätze gültiger Parameter } x_r\text{:} \\ & \{\texttt{Invariante} \wedge \mathit{pre}_r(x_r)\}\ B_r\ \{\texttt{Invariante} \wedge \mathit{post}_r\} \\ \text{2.)} & \forall \cdots \mathit{Create} \cdots \text{ und} \\ & \forall \text{ Sätze gültiger Parameter } x_{\mathit{Create}}\text{:} \\ & \{\text{Default}_C \wedge \mathit{pre}_{\mathit{Create}}(x_{\mathit{Create}})\}\ B_{\mathit{Create}}\ \{\texttt{Invariante}\} \end{cases}$$

Dabei sei: $\mathit{pre}_r(x_r)$ = „Vorbedingung von r für x_r“, B_r= „Block von r“ und post_r= „Nachbedingung von r“.

d) Schleifen:

```
from
   x:= a; y:= b
invariant                    ⟵ Korrektheit
   x > 0; y > 0;
   -- gcd(x,y) = gcd(a,b)
variant                      ⟵ Terminierung
   max(x, y)
until
   x = y
loop
   if x > y then x:= x − y
   else y:= y − x
   end
end; -- loop
```

Variante (Terminierung) $\stackrel{Def}{=}$ ganzzahliger Ausdruck, der nach Schleifenende ≥ 0 ist und bei jedem Schleifendurchgang mindestens um 1 dekrementiert wird, jedoch nie negativ wird.

e) Prädikate an beliebiger Code-Stelle:

```
check
   assertion_1;
      ⋮
   assertion_n;
end
```

Beispiel:

```
⋮
chek not s.empty end
s.pop
⋮
```

statt

```
⋮
if not s.empty
then ...
⋮
```

Check wird nur im Debugging Mode (ALL_ASSERTIONS) ausgeführt.

f) Exeptions:

Siehe Abschnitt 1.3.2, 1.3.4 und 1.3.6

`Class` *POINT*

Ancestors of class *POINT:*

GRAPH_CONST
GEN_POINT
TWO_COORD

-- Points, as implemented by X Windows.

`class interface` *POINT* `exported features`

black_color, white_color, x, y, translate, scale, visible, color, show, hide, join, distance, rotate, display, erase, set_color

`feature specification`

---

black_color: INTEGER
-- Code of black color for machine-independent results

white_color: INTEGER
-- Code of white color for machine-independent results

x: REAL

y: REAL

---

translate(d1, d2: REAL)
-- Translate by d1 horizontally, by d2 vertically.
`ensure`
$x =$ `old` $x + d1$`;`
$y =$ `old` $y + d2$

---

scale(f1, f2: REAL)
-- Scale x by f1 horizontally, y by f2 vertically.
`ensure`
$x =$ `old` $x * f1$`;`
$y =$ `old` $y * f2$

---

visible: BOOLEAN

---

color: INTEGER

---

show

-- Make point visible for next display.

`ensure`

visible

---

hide

-- Make point invisible for next display.

`ensure`

`not` *visible*

---

join (other `like` *Current; w: WINDOW; h, v: REAL)*

-- Join point to other in w (multiply coordinates by (h,v)).

`require`

argument_point_not_void: `not` *other.Void;*

window_not_void: `not` *w.Void*;

positive_scales: $h > 0.0$ `and` $v > 0.0$

`ensure`

$x =$ `old` x `and` y `=` `old` y;

other.x $=$ `old` *other.x* `and` *other.y* $=$ `old` *other.y*

---

distance(other: `like` *Current): REAL*

-- Distance between current point and point other

---

rotate (a: REAL; c: TWO_COORD)

-- Rotate point by a relative to c.

-- Angle is measured in radians

`require`

second_argument_not_void: `not` *c.Void*

---

display(w: WINDOW; h,v: REAL)

-- Display point at position $(x * h,\ y * v)$ in w.

`require`

window_not_void: `not` *w.Void*;

positive_scales: $h > 0.0$ `and` $v > 0.0$

---

erase(w: WINDOW; h,v: REAL)

-- Erase point at position $(x * h, y * v)$ from w.

`require`

window_not_void: `not` *w.Void*;

positive_scales: $h > 0.0$ `and` $v > 0.0$

---

set_color(c: INTEGER)

-- Set color to c.
-- For machine-independent results, the actual
-- argument for c
-- should be the symbolic name Black_color or
-- White_color
-- rather than a literal value.

`require`

black_or_white_color:

c = black_color `or` *c = white_color*

`ensure`

color = c

Create (x1, y1: REAL)

-- Make a point with coordinates (x1, y1).

`ensure`

visible;
color = black_color;
x = x1 `and` *y = y1*

`end interface -- class POINT`

1.6.3 Erweiterte Möglichkeiten (inkl. Quantoren) in ANNA

(ANNA = annotation language, **ann**otated **A**DA)

– **genauere Typspezifikation:** (**ADA** erlaubt auch REAL-Unterbereichstypen)

```
type TARGET is
    record
        BULLS_EYE : REAL range 1.0 .. 10.0;
        INNER_CIRCLE : REAL range 2.0 .. 20.0;
        OUTER_CIRCLE : REAL range 4.0 .. 40.0;
    end record;
--  the inner circle must be twice as wide as the bull-eye,
--  the outer circle must be twice as wide as the inner circle.
```

– **genauere Typspezifikation (ANNA):**

```
type TARGET is
    record
        BULLS_EYE : REAL range 1.0 .. 10.0;
        INNER_CIRCLE : REAL range 2.0 .. 20.0;
        OUTER_CIRCLE : REAL range 4.0 .. 40.0;
    end record;
--|     where T : TARGET =>
--|         T.INNER_CIRCLE = 2*T.BULLS_EYE and
--|         T.OUTER_CIRCLE = 2*T.INNER_CIRCLE;
```

– **Semantik von Funktionen:** (. . . durch zusätzliche „Begriffe“ (virtuelle Funktionen))

```
--: function SCORE(X, Y : REAL; T : TARGET)
--: return INTEGER;
--|    where return
--|        if X*X+Y*Y <= T.BULLS_EYE ** 2 then 10
--|        elseif X*X+Y*Y <= T.INNER_CIRCLE ** 2 then 5
--|        elseif X*X+Y*Y <= T.OUTER_CIRCLE ** 2 then 1
--|        else
--|            0
--|        end if;
```

Andere virtuelle ANNA-Funktionen zur Spezifikation etwa von

```
    procedure QUICKSORT(A : in out VECTOR);
--|    where out (A = SORTED(in A));
```

sind:

```
--: package SORTING_CONCEPTS is
--:     subtype INDEX_TYPE is INTEGER;
--:     type VECTOR is array (INDEX_TYPE range <>)
--:                                             of INTEGER;

--:     function ORDERED(A : VECTOR) return BOOLEAN;
--|         where
--|         return for all I, J : A'RANGE => I ≤ J → A(I) ≤ A(J);


--:     function PERMUTATION(A, B : VECTOR)
--:     return BOOLEAN;
--|         where
--|         in (A'LENGTH = B'LENGTH),
--|         return
--|             A'LENGTH = 0
--|                 or else
--|             (exist I : B'RANGE => A(A'FIRST) = B(I)
--|                 and
--|                 PERMUTATION(A(A'FIRST+1 .. A'LAST),
--|                             B(B'FIRST .. I−1) &
--|                             B(I+1 .. B'LAST)));


--:     function SORTED(A : VECTOR) return VECTOR;
--|         where return B : VECTOR =>
--|             PERMUTATION(A, B) and ORDERED(B);


--:     function IS_IN_INTERVAL(X : ITEM; A : VECTOR)
--:     return BOOLEAN;
--|         where
--|         return (exist I : A'RANGE => X = A(I));


--:     function PARTITIONED(A : VECTOR; I : INDEX_TYPE)
--:     return BOOLEAN;
--|         where return
--|             if I isin A'RANGE then
--|                 for all J : A'FIRST .. I => A(J) ≤ A(I) and
--|                 for all J : I .. A'LAST => A(I) ≤ A(J)
--|             else
--|                 FALSE
--|             endif


        ...             -- Other searching and sorting concepts.
--: end SORTING_CONCEPTS;
```

Die Spezifikation lautet dann etwa:

```
--: with SORTING_CONCEPTS;
--: use SORTING_CONCEPTS;
-- procedure to sort a vector A — a one dimensional array of integers.

     procedure SORT(A : in out VECTOR);
--|          where
--|               out (PERMUTATION(A, in A)),
--|               out (ORDERED(A));
```

Schrittweise Verfeinerung (d.h. Algorithmenkonstruktion) mit Annotation sieht mit diesen Hilfsmitteln folgendermaßen aus:

```
     --| with
     --|     out (ORDERED(A)),
     --|     out PERMUTATION(A, in A);
          begin
               for I in reverse A'FIRST .. A'LAST loop
                    for J in A'FIRST .. INDEX_TYPE'PRED(I) loop
                         if A(J) > A(J+1) then
                              SWAP(A, J, J+1);
                         end if;
     --|                  MAX_IN_SLICE(A(J+1), A(A'FIRST .. J+1));
                    end loop;
     --|             ORDERED(A(I .. A'LAST)) and
     --|             PARTITIONED(A, I);
               end loop;
          end;
```

wobei

```
          procedure SWAP(A : in out VECTOR; X, Y : INDEX_TYPE);
     --|     where
     --|          out (A(X) = in A(Y) and A(Y) = in A(X)),
     --|          out (for all I : A'RANGE =>
     --|               (I /= X and I /= Y => A(I) = in A(I)));
```

und

$$\texttt{MAX\_IN\_SLICE}(\texttt{X}, \texttt{A}(\texttt{I..J})) \stackrel{\text{def}}{=} \left(X := \max_{j:=I,\ldots,J} A(j) \right).$$

Ein weiteres Beispiel:

```
        procedure
             TRANSFER(AMOUNT : POSITIVE_DOLLARS_SUBTYPE;
                  OUT_ACCOUNT: in out WITHDRAWAL_SUBTYPE;
                  IN_ACCOUNT : in out DEPOSIT_SUBTYPE);
--|          where
--|               in (AMOUNT ≤ BALANCE(OUT_ACCOUNT)),
--|               out (BLANCE(OUT_ACCOUNT) =
--|                    in BALANCE(OUT_ACCOUNT)−AMOUNT),
--|               out (BLANCE(IN_ACCOUNT) =
--|                    in BALANCE(IN_ACCOUNT)+AMOUNT),
--|               out (for all A : ACCOUNT_TYPE =>
--|                 A /= OUT_ACCOUNT and A /= IN_ACCOUNT →
--|                    BALANCE(A) = in BALANCE(A));
```

– **implizite Typspezifikation:**

```
     declare
          X : INTEGER range 1..10000;
--|       X mod 2 = 0;
```

Beispiele für quantifizierte Spezifikationen:

```
     type DAY is (SUN, MON, TUE, WED, THU, FRI, SAT);
--     Every value of DAY has length 3.
--|    for all X : DAY => DAY'IMAGE(X)'LENGTH = 3;
--     Names of weekdays do not contain 'S'.
--|    for all X : DAY range MON .. FRI =>
--|         for all I : 1 .. DAY'WIDTH =>
--|                   DAY'IMAGE(X)(I) /= 'S';
```

oder kürzer:

```
--|    for all X : DAY range MON .. FRI;
--|         I : 1 .. DAY'WIDTH => DAY'IMAGE(X)(I) /= 'S';
```

– **Initialisierung:**

```
     type SQUARE_MATRIX is array (1 .. N, 1 .. N) of REAL;
     A : SQUARE_MATRIX;
          ...
     for I in 1 .. N loop
          A(I) := (1 .. I−1 => 0, others => 1);
     end loop;
--|     A'DEFINED,
--          Test that all components of A are initialized.
--|     TRIANGULAR(A);
--          A call to a function testing if A is triangular.
```

– **Schleifeninvarianten:**

```
--     These annotations constrain execution of the loop body.
--| with
--|     in LOW ≤ LOW and HIGH ≤ in HIGH,
--|     LOW ≤ MID+1 and MID ≤ HIGH,
--|     ORDERED(A(in LOW .. in HIGH)),
--|     IN_INTERVAL(X, A (in LOW .. in HIGH))
--|                 → IN_INTERVAL(X, A(LOW .. HIGH));
       while LOW < HIGH loop
            MID := (LOW+HIGH)/2;
            if X > A(MID) then
                 LOW := MID+1;
            else
                 HIGH := MID;
            end if;
       end loop;
```

– **Ergebnis-Spezifikation:**

```
     function SQUARE_ROOT(N : NATURAL) return NATURAL;
--|     where
--|          N ≥ 0;
--|          return S : NATURAL => S ** 2 ≤ N < (S+1) ** 2;
--     Read "return S natural such that ... "

     function FACTORIAL(N : NATURAL) return POSITIVE;
--|     where return
--|       (if N=0 then 1 else N*FACTORIAL(N−1) end if);
```

– **semantisch spezifizierte Untertypen:**

```
        subtype LETTERS is CHARACTER;
--|        where S : LETTER =>
--|            S isin CHARACTER range 'A' .. 'Z' or
--|            S isin CHARACTER range 'a' .. 'z';
```

bzw.

```
        type INTERVAL is
            record
                LEFT_POINT, RIGHT_POINT : REAL;
            end record;
--|         where X : INTERVAL =>
--|                 X.LEFT_POINT ≤ X.RIGHT_POINT;
```

– **Semantik eines ADT Stack:**

```
        package STACK_MANAGER is

            type STACK is private;
--:         function LENGTH(S : STACK) return NATURAL;
--:         function TOP(S : STACK) return ITEM;

            procedure PUSH(X : in ITEM; S : in out STACK);
--|            where out (LENGHT(S) = LENGTH(in S)+1),
--|                  out (TOP(S) = X);

            procedure POP(Y : out ITEM; S in out STACK);
--|            where out (LENGHT(S) = LENGTH(in S)−1),
--|                  out (Y = TOP(in S));

        private ...

        end STACK_MANAGER;
```

– **dürftige numerische Spezifikationen:** [10]

Spezifikation von SIN:

```
--: function SINE_SERIES(X : FLOAT; CUT_OFF : INTEGER)
--: return FLOAT is
--:     Y : FLOAT := 0.0;
--: begin
--:     for I in 0 .. CUT_OFF loop
--:         Y := Y+FLOAT((−1) ** I) /
--:             FLOAT(FACTORIAL(2*I+1))*X ** (2*I+1);
--:     end loop;
--:     return Y;
--: end SINE_SERIES;

-- Sine function to be implemented by a more efficient algorithm.

  function SIN(X : RADIANS_TYPE) return AMPLITUDE_TYPE;
--|     where return A : AMPLITUDE_TYPE =>
--|         abs (FLOAT(A)−SINE_SERIES(X, CUT)) <
--|                         MAX_DELTA;
```

[10] Eine zukünftige Verbesserung in diesem Bereich ist wegen der Existenz von IEEE 754-1985 und LIA zu erwarten.

Kapitel 2

Evaluation von Software: Gütekriterien und deren Überprüfbarkeit

Probleme bei Software ,,geringer" Qualität, etwa am Beispiel von THERAC 25[1]:

> Selten sind solche schädlichen Vorfälle so gut dokumentiert worden wie im Fall des ,,THERAC 25", eines computergestützten Bestrahlungsgerät, welches in zwei ,,Modi" arbeitet: im ,,X-Modus" wird ein Elektronenstrahl von 25 Millionen Elektronen-Volt durch Beschuß einer Wolframscheibe in Röntgenstrahlen verwandelt; im ,,E-Modus" werden die Elektronen selbst, allerdings ,,weicher" mit erheblich reduzierter Energie als Korpuskelstrahlung erzeugt. Je nach therapeutischer Indikation wird die geeignete Strahlungsart eingestellt; in beiden Fällen kann der Bestrahlungsverlauf, nach Modus, Dauer, Intensität und Bewegungskurve der Strahlungsquelle, mit einem Bildschirm-,,Menü" eingegeben werden.
>
> Als mehrere Patienten berichteten, sie hätten bei Behandlungsbeginn das Gefühl gehabt, ,,ein heißer Strahl durchdringe" sie, wurde dies vom Hersteller als ,,unmöglich" zurückgewiesen. Erst nach dem Tode zweier Patienten sowie massiven Verbrennungen bei weiteren Personen kam heraus, daß neben dem X- und E-Modus mit niedriger Elektronenintensität infolge Programmierfehler ein unzulässiger dritter Zustand auftrat, nämlich direkt wirkende, 25 Millionen Elektronen-Volt ,,heiße" Elektronen. Dies geschah immer dann, wenn ein vorgegebenes ,,Behandlungsmenü" mittels Cursor-Taste modifiziert wurde. Um aufwendige Umprogrammierung zu vermeiden, wollte der kanadische Hersteller die Benutzung der Cursor-Tasten verbieten bzw. diese ausbauen und die Tastenlücke mit Klebeband abdichten lassen! Es ist zu befürchten, daß der Fall ,,THERAC 25" kein Einzelfall ist. Zumeist ist es mangels entsprechender Vorsorge in computergesteuerten Medizingeräten schwerlich möglich, schädliche Systemverhalten später aufzuklären.

[1] Zeitungszitat

2.1 Orange-Book

(Department of Defence Trusted Sytem Evaluation Criteria, DOD 5200.28-STD, December, 1985)

Das Orange-Book bezieht sich auf Absicherung gegenüber

- unbefugten Informationsgewinns und
- unbefugter Modifikation von Informationen

nicht jedoch

- unbefugter Funktionalitätsbeeinträchtigung[2].

Es existieren die folgenden Evaluations-Klassen:

Klasse	Name	Beispielsysteme
D	Minimale Sicherheit	Keine. Falls IBM PC's mit MS-DOS, Apple Macintosh-Systeme u.ä. zur Evaluation eingereicht würden, so fielen sie in diese Klasse.
C1	Sicherheit nach eigenem Ermessen	IBM VMS/RACF, „normales" UNIX, falls es zur Evaluation eingerichtet würde.
C2	Sicherheit durch Zugriffskontrolle	DEC, VAX/VMS 4.3, Sun Solaris 2, ...
B1	ausgewiesene Sicherheit	AT&T System V/MLS, IBM MVS/ESA, ...
B2	in Systemstruktur integrierte Sicherheit	Tusted XENIX, ...
B3	geschlossenes Sicherheitssystem	XTS-200, ...
A1	verifiziertes Design	SCOMP, Boeing SNS, ...

Diese Klassen werden je nach

- Sicherheitsmechanismen,
- Zugriffskontrolle,
- Betriebsqualität und
- Dokumentation-Anforderungen

definiert.

[2]und erst recht nicht gegen „nicht implementierte Funktionalität"

Eine Übersicht der Anforderungen bzw. der von Klasse zu Klasse auftretenden Verschärfungen der Anforderungen ist aus der folgenden Aufstellung abzulesen[3]:

Trusted Computer System Evaluation Criteria Summary Chart

	C1	C2	B1	B2	B3	A1	
Discretionary Access Control	2	2	3	3	2	3	Security Policiy
Objekt Reuse	1	2	3	3	3	3	
Labels	1	1	2	2	3	3	
Label Integrity	1	1	2	3	3	3	
Exportation of Labeled Information	1	1	2	3	3	3	
Exportation of Multilevel Devices	1	1	2	3	3	3	
Exportation of Single-Level Devices	1	1	2	3	3	3	
Labeling Human-Readable Output	1	1	2	3	3	3	
Mandatory Access Control	1	1	2	2	3	3	
Subject Sensitivity Labels	1	1	1	2	3	3	
Device Labels	1	1	1	2	3	3	
Indetification and Authentication	2	2	2	3	3	3	Accountability
Audit	1	2	2	2	2	3	
Trusted Path	1	1	1	2	2	3	
System Archtecture	2	2	2	2	2	3	Assurance
System Integrity	2	3	3	3	3	3	
Security Testing	2	2	2	2	2	2	
Design Specification and Verfikation	1	1	2	2	2	2	
Covert Channel Analysis	1	1	1	2	2	2	
Trusted Facility Management	1	1	1	2	2	3	
Configuration Management	1	1	1	2	3	2	
Trusted Recovery	1	1	1	1	2	3	
Trusted Distribution	1	1	1	1	1	2	
Security Features User's Guide	2	3	3	3	3	3	Documentation
Trusted Facility Manual	2	2	2	2	2	3	
Test Documentation	2	3	3	2	3	2	
Design Documentation	2	3	2	2	2	2	

1 No requirements for this class

2 New or enhanced requirements for this class

3 No additional requirements for this class

[3]Genaueres ist etwa in [34] nachzulesen.

C1-Systeme:

enthalten eben genug Sicherheitmechanismen, um in einer kooperativen DV-Umwelt (etwa Multiuser oder Workgroups) den Benutzer davor zu schützen, ernsten Schaden anzurichten, der das System beschädigen oder mit der Arbeit anderer interferieren könnte: **Paßworte** und **benutzerdefinierte Dateizugriffsrechte, Prozeßzugriffsrechte,**... C1-Systeme sind auf keine Weise vor Infiltration geschützt.

C2-Systeme:

müssen zusätzlich

- ein **Accounting** einzelner Benutzer zulassen (,,wer tut wann was"),
- **spezifischeren Dateizugriffsschutz** erlauben (nur Nutzer „Hans" und „Bertold" dürfen auf die Datei „xyz" zugreifen),
- zufällig im System verbleibende Daten vor dem Zugriff anderer Benutzer schützen.

Die Test- und Dokumentationsanforderungen sind ebenfalls größer.

Eine neuere Klasse von „sicheren Systemem" sind die **CMW's** (= compartmented mode workstations), die als sichere Rechner und nicht nur als Terminals in einem Netz betrieben werden können; so kann ein Benutzer TOP SECRET-Resourcen des Projekts ALPHA nutzen, ohne auch TOP SECRET-Rechte bzgl. Projekt OMEGA besitzen zu müssen (der Schutz geht bis auf die Operationen „cut and paste" herunter).

2.2 IT-Sicherheitskriterien (ITSEC)

Ähnlich wie das Verteidigungsministerium der USA hat die ZSI (=Zentralstelle für Sicherheit für das Chiffrierwesen) im Bundesanzeiger 1989 bzw. 1992 eine eigene Sicherheitsklassifizierung definiert, die sich sogar auch gegen „unbefugte Funktionalitätsbeeinträchtigung“ wendet:

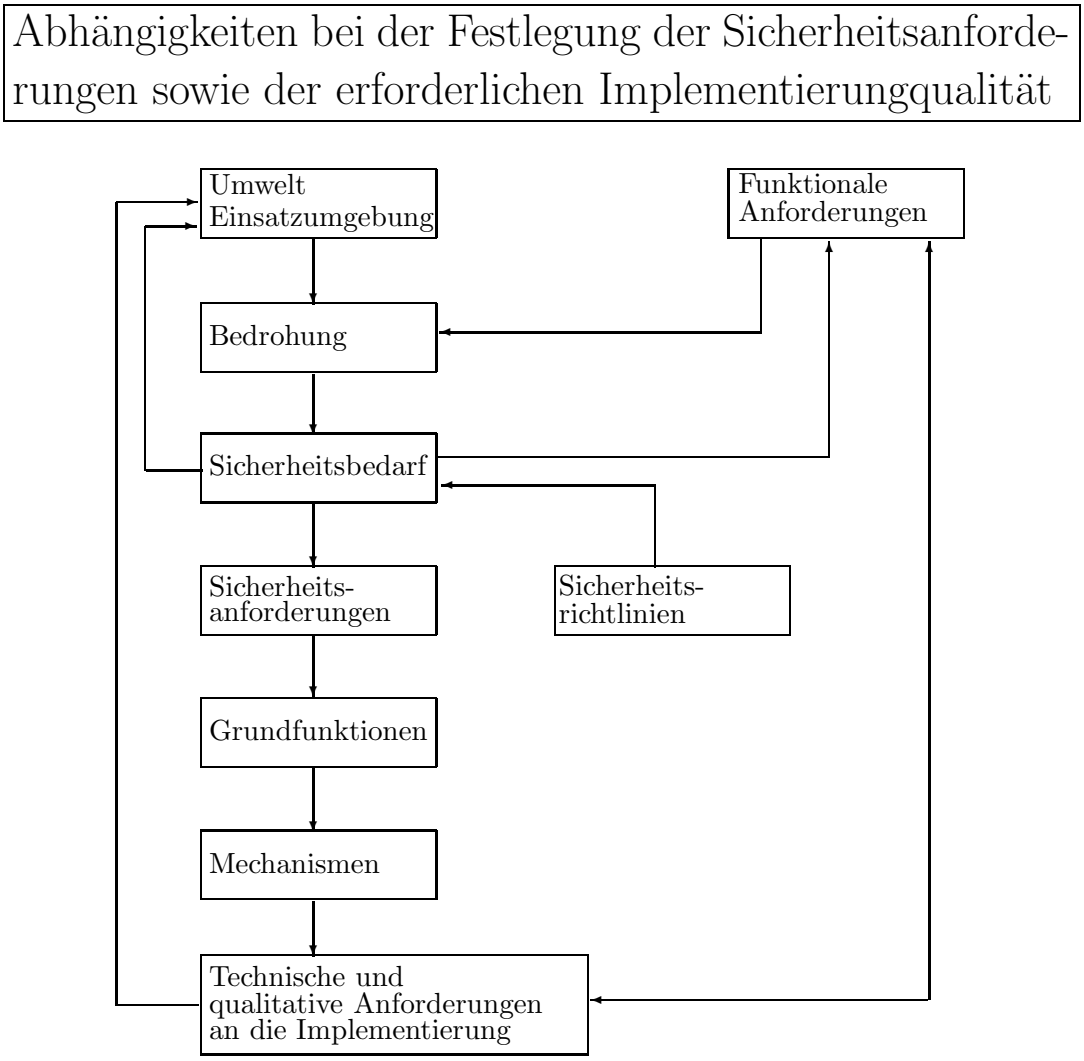


Abbildung 2.1: Abhängigkeiten von Sicherheitsanforderungen

Die Grundfunktionen sicherer Systeme:

1) Indentifikation und Authentisierung,
2) Rechteverwaltung,
3) Rechteprüfung,
4) Beweissicherung,
5) Wiederaufbereitung,
6) Fehlerüberbrückung,
7) Gewährleistung der Funktionalität und
8) Übertragungssicherung

sollen dabei gegen die drei Grundbedrohungen

1) Verlust der Vertraulichkeit,
2) Verlust der Integrität und
3) Verlust der Verfügbarkeit

durch die (geforderte) Güte der Mechanismen 1 bis 8 resistent gemacht werden.

Die Zertifizierung von Software in Funktionalitätsklassen

F1	F2	F3	F4	F5		
2	2	3	2	3	Subjektidentifizierung und -authentisierung	Identifikation und Authentisierung
3	1	1	2	2	Vertrauenswürdiger Zugriffspfad	
2	2	3	2	2	Benutzerbestimmbarer Zugriffsschutz	Rechte-verwaltung
1	1	2	3	3	Festgelegter, regelbasierter Zugriffsschutz	
1	1	2	3	3	Attribute für Subjekte und Objekte	
1	1	2	3	3	Integrität der Attribute	
1	1	1	2	3	Ausgabe von Inforamtionen über Attributwerte	
1	1	2	3	3	Ausgabe über einstufige Kanäle	
1	1	2	2	3	Ausgabe über mehrstufige Kanäle	
1	1	2	3	3	Ausgabe von attributierten Informationen	
1	1	2	3	3	Attribute menschenlesbarer Ausgaben	
2	3	2	3	3	Überprüfung von Ereignissen und Daten	Rechteprüfung
1	2	2	2	3	Protokollierung von Ereignissen und Daten	Beweis-sicherung
1	2	3	3	3	Auswertung und Dokumentation von Protokolldaten	
1	1	1	1	2	Überwachung sicherheitskritischer Ereignisse	
1	2	3	3	3	Wiederaufbereitung von Objekten	Wiederaufbereitung

1 keine Anforderungen an diese Klasse

2 neue oder erweiterte Anforderungen an diese Klasse

3 keine Zusatzanforderungen

Erweiterungen 1992:

Klasse		Beispiel
F6	Eine eigene Klasse für Systeme mit hohen Integritäts- (statt Vertraulichkeits-) Anforderungen	Datenbanksysteme
F7	Eine eigene Klasse für Systeme mit hohen Anforderungen für ein komplettes System oder eine spezielle Funktion eines Systems	Prozeßkontrollsysteme
F8	Eine eigene Klasse für Systeme mit hohen Anorderungen an den Sicherheitsschutz der Datenintegrität während der Datenkommunikation	
F9	Eine eigene Klasse für Systeme mit hohen Anforderungen an die Vertraulichkeit von Daten während der Datenkommunikation	kryptographische Systeme
F10	Klassen, die F8 und F9 erfüllen	Schutz relevanter Daten auf öffentlichen Netzen

und Qualitätsklassen

Q1	Q2	Q3	Q4	Q5	Q6	Q7		
2	2	3	2	3	3	3	Spezifikation der Sicherheitsanforderungen	Qualität der Sicherheits-anforderungen
1	1	1	1	2	2	2	Formales Sicherheitsmodell	
2	2	2	2	3	3	3	Verbale Entwurfsspezifikation	Qualität der Spezifikation
1	1	1	1	2	3	3	Semiformale Entwurfsspezifikation	
1	1	1	1	1	2	2	Formale Entwurfsspezifikation	
1	1	1	1	2	2	3	Werkzeuge zur Prüfung der Spezifikation	
2	2	2	2	2	2	2	Implementierte Mechanismen und Algorithmen	Qualität der Mechanismen
2	3	3	3	3	3	3	Spezifikation der Abgrenzung	Qualität der Abgrenzung
2	3	3	2	3	3	2	Abgrenzungsmechanismen	
1	1	1	1	2	2	2	Protok. u. Analyse von verdeckten Kanälen	
1	2	2	2	2	2	2	Abbildbarkeit Spezifikation ↔ Implementierung	Qualität des Herstellungs-vorgangs
1	2	3	3	3	3	3	Bibliothek mit Testprogrammen	
1	1	2	2	2	2	2	Implementierungssprachen und Tools	
1	1	2	3	2	2	3	Änderungs- und Versionskontrolle	
1	1	2	3	2	2	3	Softwareintegrations- u. Abnahmeverfahren	
1	1	1	1	1	1	2	Vertrauenswürdiges Entwicklungspersonal	
1	2	2	3	3	3	3	Vertrauenswürdige Systemgenerierung	Betriebs-qualität
1	2	3	3	3	3	3	Vertrauenswürdige Systemeinspielung	
1	1	1	1	2	3	3	Vertrauenswürdige Systemverteilung	
1	1	2	3	2	3	3	Vertrauenswürdiger Systemstart	
1	1	1	1	2	3	3	Vertrauenswürdiger Wiederanlauf	
2	2	3	3	2	2	3	Systemkonfigurierung	
1	2	3	3	2	2	3	Wartung der Hardware und Software	
1	2	3	3	2	3	3	Selbsttests der Hardware	
2	3	3	3	3	3	3	Anwenderdokumentation der Sicherheitsfunktionen	Qualität der Dokumentation

1 keine Anforderungen an diese Klasse

2 neue oder erweiterte Anforderungen an diese Klasse

3 keine Zusatzanforderungen

Q0	unzureichende Qualität
Q1	getestet[1]
Q2	methodisch getestet[1]
Q3	methodisch getestet und teilanalysiert [1]
Q4	informell analysiert[1]
Q5	semiformal analysiert[1]
Q6	formal analysiert[1]
Q7	formal verifiziert[1]

geschieht dabei durch die Arbeit eines Evaluations-Teams, bestehend aus:

1. dem organisatorischen Projektleiter (Projektverantwortlicher),
2. dem technischen Projektleiter,
3. den Evaluatoren und
4. dem Moderator.

Die Zuständigkeit dieser bei einer bundeszentralen Stelle lokalisierten Teams gestalltet sich dabei wie folgt:

1. Der organisatorische Projektleiter ist verantwortlich für den gesamten Ablauf und die Durchführung der Evaluation. Er stellt sicher, daß der Zeitplan eingehalten wird und die Kosten der Evaluation im vorgegebenen Rahmen bleiben.

Er hat folgende Aufgaben zu erfüllen:

- Vertragsgestaltung (Festlegung des zu evalutierenden Produkts, der angestrebten Funktionalitätsklasse und der Qualitätsstufe mit dem Hersteller),
- Mitarbeit an der Evaluationsplanung,
- Teilnahme an Reviews während der Evaluation,
- Berichterstattung.

2. Der technische Projektleiter ist verantwortlich für den technischen Ablauf der Evaluation. Er verteilt die Aufgaben an die Evaluatoren.

Er hat folgende Aufgaben zu erfüllen:

- Projektplannung (Festlegng des technischen Ablaufs der Evaluation und gegebenenfalls Änderungen des Projektplans),
- aktive Teilnahme an der Evaluation,
- Berichterstattung,
- Entscheidung über technische Probleme.

[1] der Sicherheitsfunktionen

3. **Die Evaluatoren** führen die Evaluation durch.

   Sie haben folgende Aufgaben zu erfüllen:

   - Beurteilung der einzelnen Dokumente,
   - Entwicklung von Testprogrammen,
   - Erstellung von Testdatensätzen,
   - Durchführung von Tests,
   - Beurteilung der Anwender-Dokumentation,
   - Erarbeitung von Dokumenten zu allen Evaluationsvorgängen,
   - Berichte über auftretene Probleme und vorzeitige Beendigung einer Teilevaluation (an den technischen Projektleiter),
   - Präsentation und Begründung der Prüfergebnisse von Teilevaluationen bei Reviewsitzngen.

4. **Der Moderator** ist zuständig für die Planung und Durchführung von Reviews.

   Er hat folgende Aufgaben zu erfüllen:

   - Planung von Reviewsitzungen (Festlegung von Zeit und Ort, Benennung und Benachrichtigung teilnehmender Personen, Vorbereitung des Reviews, Verteilung der Arbeitsunterlagen),
   - Moderation von Reviewsitzungen,
   - Berichterstattung.

Das auszustellende Zertifikat ist das Abschlußdokument der Evaluation. Es besteht aus den folgenden drei Teilen:

1. Zertifikat selbst (öffentliches Dokument, das grundsätzliche Aussagen über das evaluierte System beinhaltet),
2. Anhang 1 (öffentliches Dokument, das detaillierte Angaben zum evaluierten System enthält),
3. Anhang 2 (nicht öffentliches Dokument, das nur den Hersteller und die Evaluationsbehörde bestimmt ist).

Der Vorgang der Evaluation in einer Übersicht:

Phase	**Stufe**
1 Kontaktaufnahme	1. Antrag auf Evaluation
	2. Bearbeitung des Evaluationsantrags
	3. Festlegung der Evaluationskriterien
	4. Ausarbeitung der Evaluationsverträge
2 Dokumentprüfung	1. Ausarbeitung eines ersten Arbeitsplanes
	2. Ist die Durchführung der Evaluation möglich?
	3. Erstellung eines Gesamtdokuments aus Stufe 2
	4. Übergabe des Dokumentes an Auftraggeber
	5. Stufe 1 bis 3 noch einmal durchlaufen
	6. Entgültige Entscheidung über Durchführung
3 Inhaltliche Dokumenten- und Objektprüfung	1. Installation des zu prüfenden Objekts
	2. Erstellung des detaillierten Arbeitsplans
	3. Begin der Evaluation
	4. Projektfortschrittsüberprüfung
	5. Evaluaton gemäß Arbeitsplan
	6. Auswertung der Einzelergebnisse
4 Vorarbeiten für die Zertifikaterstellung	1. Erstellung eines internen Dokuments
	2. Erstellung eines Evaluationsberichts
	3. Archivierung aller Ergebnisse
5 Erstellen Zertifikat	1. Dokumente an Evaluationsbehörde übergeben
	2. Prüfung der Evaluationsergebnisse
	3. Erstellung des Zertifikates

Wichtig sind hier (wie auch schon bei LIA) die Anforderungen an eine ausreichende Dokumentation der Sicherheitsfunktionen der zu evaluierenden Software, für die Qualitätsstufen Q1,. . . ,Q3 etwa:

1. Beschreibung der Sicherheitsanforderungen,
2. Spezifikation der zu evaluierenden Systemteile,
3. Beschreibung der Abgrenzung zu nicht zu evaluierendn Systemteilen und der Schnittstelle zu diesen Teilen,
4. Dokumentation für den Anwender, d.h.
   - Beschreibung der Anwendung der Sicherheitsfunktionen, aufgeteilt nach den in den Sicherheitsanforderungen festgelegten Rollen,
   - Beschreibung der sicherheitsrelevanten Aspekte bei Systemgenerierung, Systemstart, Systemverwaltung und Systemwartung,

5. Beschreibung der verwendeten Hard- und Firmware mit Darlegung der Funktionalität der in Hardware bzw. Firmware realisierten Schutzmechanismen,

6. Testdocumentation (ab Q2).

Bemerkung: Beziehungen zwischen dem Orange-Book und ITSEC:

IT-Sicherheitskatalog	Orange-Book
Q0	D
F1, Q2	C1
F2, Q2	C2
F3, Q3	B1
F4, Q4	B2
F5, Q5	B3
F5, Q6	A1
Q7	besser als A1

Hauptschwerpunkt sowohl des Orange-Book als auch der ITSEC ist die Qualität der Sicherheit von „Daten“. Für die SW-Güte an sich ist jedoch ein ähnliches Klassifizieren der Funktionalität der SW nötig, das in der höchsten Stufe „formale Spezifikation“ und „formale Verifikation“ der Quellen der Software beinhalten sollte. Diese höchste Stufe sollte per Gesetz für gewisse Softwareklassen – wie etwa dem am Kapitelanfang geschilderten System THERAC 25 – vorgeschrieben werden.

Anhang A

Übungen

A.1 Übungsblatt 1 — Spezifikation

Aufgabe 1. *Hoare-Tripel*

a.) Geben Sie bitte das Hoare-Tripel zum Statement

```
begin
     r := 0;
     for i := 1 to n do begin
          r:= 10 * r + d[i];
          d[i] := r div 2;
          r := r - 2 * d[i];
     end;
end
```

an.

b.) Was ist das Ziel des Statements?

c.) Wie muß d aussehen? Welche semantische Bedeutung hat d?

Aufgabe 2. *Spezifikation*

Spezifizieren Sie das Statement aus Aufgabe 1 durch die Angabe von

- Definitionsbereich
- Wertebereich
- funktionalem Zusammenhang von Eingabe- und Ausgabegrößen
- Eigenschaften der Lösung, Vorbedingungen, Nachbedingungen, Nebeneffekte, ...

Aufgabe 3. *Verifikation*

a.) Verifizieren Sie das Erreichen der Spezifikation durch Angabe weiterer Annotationen zwischen den einzelnen Zeilen des Statements von Aufgabe 1. Vergessen Sie die Invariante bzw. die Invarianten nicht.

b.) Zeigen Sie den endlichen Abbruch des Statements durch Angabe einer Variante. (Bitte auch als Annotation im Programmtext vermerken!)

Aufgabe 4. *IEEE 754 Floating Point System, Rundung*

Gibt es bei der Rundung „to nearest“ Unterschiede, wenn die Basis 2 beziehungsweise die Basis 16 benutzt wird? (Begründung oder Beispiele)

Aufgabe 5. *Floating Point System*

Wie ist die Spezifikation

$$\forall\ x \in \mathbb{R} \text{ berechne } \sin(x) \in \mathbb{R} \cap [-1.0, +1.0]$$

einzuschätzen? Geben Sie gegebenenfalls eine „bessere“ an.

A.2 Übungsblatt 2 — Spezifikation und informelle Verifikation

Aufgabe 1. *readreal*

Diskutieren Sie das Unterprogramm *readreal* analog wie das Unterprogramm *write-real* in der Vorlesung diskutiert wurde:

```
procedure readreal(var f: text; var x:real);
   {Einlesen einer reellen Zahl x vom File f}
   {folgende Konstanten sind systemabhängig }
   const t48 = 281474976710656;      {48 Bit Mantisse}
         limit = 56294995342131;     {= t48 div 5}
         z = 27;                     {= ord('0') }
         lim1 = 322;                 {grösster Exponent}
         lim2 = -292;                {kleinster Exponent}
   type posint = 0..323;
   var ch: char; y: real; a,i,e: integer;
       s,ss: boolean;                {Vorzeichen}

   function ten(e: posint):real;     { = 10**e, 0<e<322 }
      var i: integer; t: real;
      begin i := 0; t := 1.0;
         repeat if odd(e) then
            case i of
             0: t := t * 1.0e1;
             1: t := t * 1.0e2;
             2: t := t * 1.0e4;
             3: t := t * 1.0e8;
             4: t := t * 1.0e16;
             5: t := t * 1.0e32;
             6: t := t * 1.0e64;
             7: t := t * 1.0e128;
             8: t := t * 1.0e256;
            end ;
            e := e div 2; i := i+1;
         until e = 0;
         ten := t
      end ;

   begin
      if eof(f) then
      begin message(' tried to read past end of file f');
            halt
```

```
end;
{überspringe führende Leerzeichen}
while (f^ = ' ') and  (not eof(f)) do get(f);
if not eof(f) then
begin
   ch := f^;
   if ch = '-' then
      begin s := true; get (f); ch := f^;
      end else
      begin s := false;
         if ch = '+' then
         begin get (f); ch := f^
         end
      end ;
   if not (ch in ['0'..'9']) then
   begin message('  digit expected '); halt;
   end;
   a := 0; e := 0;
   repeat if a < limit then a := 10*a + ord(ch)-z
                       else e := e+1;
      get(f); ch := f^
   until not (ch in ['0'..'9']);
   if ch = '.' then
   begin { Einlesen des Bruches } get(f); ch := f^;
      while ch in ['0'..'9'] do
      begin if a < limit then
            begin a := 10*a + ord(ch)-z; e := e-1
            end ;
         get(f); ch := f^
      end
   end ;
   if ch = 'e' then
   begin { Einlesen des Skalierungsfaktors }
      i := 0; get(f); ch := f^;
      if ch = '-' then
      begin ss := true; get(f); ch := f^
      end else
      begin ss := false; if ch = '+' then
         begin get(f); ch := f^
         end
      end ;
      if ch in ['0'..'9'] then
      begin i := ord(ch)-z; get(f); ch := f^;
         while ch in ['0'..'9'] do
         begin if i < limit then i := 10*i + ord(ch)-z;
```

```
                get(f); ch := f^
             end
          end else
          begin message('  digit expected '); halt
          end ;
          if ss then e := e-i else e := e+i;
       end ;
       if e < lim2 then
          begin a := 0; e := 0
          end else
       if e > lim1 then
       begin message(' number too large '); halt end;
       { 0 < a < 2**49 }

       if a >= t48 then y := ((a+1) div 2) * 2.0 else y := a;
       if s then y := -y;
       if e < 0 then x := y/ten(-e) else
       if e <> 0 then x := y *ten(e) else x := y;
    end;
end  { readreal }
```

a.) Kritisieren Sie den Programmierstil des Unterprogramms *readreal* (hardcoded, Konstantenbenutzung, ...).

b.) Modifizieren Sie die Funktion *ten* für das IEEE single precision float format und spezifizieren Sie diese Version "'vollständig"' (insbesondere Definitionsbereich).

c.) Verifizieren Sie Ihre Version von *ten* aus Aufgabenteil b.

d.) Versuchen Sie ansatzweise eine Spezifikation von *readreal* zu schreiben.

Aufgabe 2. *Spezifikation und Implementierung*

Schreiben Sie eine Spezifikation und eine Implementierung einer Funktion, die den *arithmetischen Mittelwert zweier Argumente* berechnen soll.

A.3 Übungsblatt 3 — Varianten eines Algorithmus in der Spezifikation

Aufgabe 1. *Spezifikation des arithmetischen Mittelwerts (Version A)*

Es werde der arithmetische Mittelwert durch

$$Z := X + (Y - X)/2$$

berechnet.

Spezifizieren Sie durch

a.) Definitionsbereich (genau!)

b.) Wertebereich

c.) Definition des Resultats

d.) Eigenschaften des Resultats

e.) Vor- und Nachbedingungen, Nebeneffekte

Aufgabe 2. *Spezifikation des arithmetischen Mittelwerts (Version B)*

Es werde der arithmetische Mittelwert durch

$$Z := (X + Y)/2$$

berechnet.

Spezifizieren Sie durch

a.) Definitionsbereich (genau!)

b.) Wertebereich

c.) Definition des Resultats

d.) Eigenschaften des Resultats

e.) Vor- und Nachbedingungen, Nebeneffekte

Aufgabe 3. *arithmetischer Mittelwert*

```
function arithMittel( x, y : real ) : real;
begin
     if (sign(x) = sign(y)) then
          arithMittel := x + (y - x)/2
     else
          arithMittel := (x + y)/2;
end { arithMittel }
```

Spezifizieren Sie durch

a.) Definitionsbereich (genau!)

b.) Wertebereich

c.) Definition des Resultats

d.) Eigenschaften des Resultats

e.) Vor- und Nachbedingungen, Nebeneffekte

Aufgabe 4. *Eigenschaften von* `arithMittel`

Sind alle der wünschenswerten Eigenschaften

1. *arithMittel* soll höchstens ein ULP falsch sein (außer evtl. im Underflow-Bereich).
2. *arithMittel* soll immer zwischen x und y liegen.
3. Falls zwischen x und y mindestens ein weiterer Rasterpunkt des Gleitkommasystems liegt, so soll *arithMittel* streng zwischen x und y liegen.
4. Es soll nie ein Overflow auftreten können.
5. Underflow soll nur in dem Falle auftreten dürfen, wenn das (exakte) Ergebnis im Absolutwert kleiner als `FLT_MIN` ist.

erfüllt?

Aufgabe 5. *Fortsetzung von Aufgabe 2*

Wo kann im Falle $r > 2$ die Eigenschaft 2 von Aufgabe 4 verletzt sein. Geben Sie ein Beispiel für $r = 10$ und 4 signifikante Ziffern an. (Hinweis: Wählen Sie zwei Argumente, die sich im Exponenten um Eins unterscheiden.)

A.4 Übungsblatt 4 — Verifikation und Codeoptimierung

Aufgabe 1. *Rundung und implizite erhöhte Genauigkeit*

Sei $A := 12.34, B := 89.08, C := 23.45, D := -45.58$. Dann ist

$$A \cdot B + C \cdot D = 30.3962 \approx 30.40.$$

Bei Benutzung einer 4-stelligen Dezimalarithmetik sei

$$R_m := rnd_m(A \cdot B),\ S_n := rnd_n(C \cdot D),\ T_{mn} := rnd_4(R_m + S_n).$$

a.) Berechne $R_4,\ S_4,\ T_{44}$.

b.) Berechne und stelle in einer Tabelle dar:

Environment 2	$S_4 =$	$S_5 =$
$R_4 =$	$T_{44} =$	$T_{45} =$
$R_5 =$	$T_{54} =$	$T_{55} =$

c.) Fülle auch die folgende Tabelle aus:

Environment 3	$S_4 =$	$S_8 =$
$R_4 =$	$T_{44} =$	$T_{48} =$
$R_8 =$	$T_{84} =$	

d.) Ergänze schließlich auch:

Environment 3	$S_4 =$	$S_8 =$
$R_4 =$	$T_{44} =$	$T_{48} =$
$R_8 =$	$T_{84} =$	$T_{88} =$

Alle T_{ij} sind Näherungslösungen für $A \cdot B + C \cdot D$. T_{ij} und T_{ji} sind „kommutativ“ zueinander berechnet worden.

e.) Gilt das Kommutativgesetz?

f.) $R_4,\ S_4$ und T_{44} gelten bei reiner 4-stelliger Dezimalarithmetik als Zwischen- bzw. Endergebnis (siehe a.)). Welche erhöhte Zwischenergebnis-Genauigkeiten existieren bei b.), c.) und d.)?

Aufgabe 2. *Verifizieren*

Verifizieren Sie die Funktion `CABS`, nachdem sie sie zuvor spezifiziert haben:

```
REAL FUNCTION CABS (Z)

     COMPLEX Z
     REAL S,ZI,ZR
     INTEGER I      ! Flags overflow iff equal -1
     INTRINSIC REAL, AIMAG, SQRT, ABS, MAX, CONDITION_SET

     I = 0

     ENABLE (OVERFLOW, UNDERFLOW)
          ZR = REAL(Z)
          ZI = AIMAG(Z)
          CABS = SQRT(ZR**2 + ZI**2)
     HANDLE (OVERFLOW, UNDERFLOW)

          ENABLE (OVERFLOW, UNDERFLOW)
               S = MAX(ABS(ZR), ABS(ZI))
               CABS = S*SQRT( (ZR/S)**2 + (ZI/S)**2 )
          HANDLE (OVERFLOW)
               I = -1
          HANDLE (UNDERFLOW)
               CABS = S
          END ENABLE

     END ENABLE

     IF (I == -1) CALL CONDITION_SET('OVERFLOW', I)

END FUNCTION CABS
```

Aufgabe 3. *Verifikation und Codeoptimierung*

Die drei Varianten zur Berechnung von x^n gemäß

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do
          if odd(n) then begin
               Teilerg := Teilerg * x; n := n - 1;
          end else begin
               x := sqr(x); n := n div 2;
          end;
          ⋮
```

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do begin
          while not odd(n) do begin
               x := sqr(x); n := n div 2;
          end;                                    { n ungerade}
          Teilerg := Teilerg * x; n := n - 1;
     end;
     ⋮
```

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do begin
          if odd(n) then begin
                Teilerg := Teilerg * x; n := n - 1;
          end;                                          { n gerade}
          x := sqr(x); n := n div 2;
     end;
     ⋮
```

berechnen bei Erfolg alle die Potenz x^n.

a.) Spezifizieren Sie die Varianten 1, 2 und 3 (vgl. Hinweise der Vorlesung).

b.) Verifizieren Sie alle 3 Varianten.

c.) Die Anweisung `n := n - 1` kann eventuell entfallen, da nach jedem ungeraden Fall sicher ein gerader Fall mit der Anweisung `n := n div 2` folgt. In welchen der drei Varianten funktioniert das? Verifizieren Sie!

A.5 Übungsblatt 5 — Verifikation rekursiver Algorithmen

Aufgabe 1. *„Qualität" von Spezifikationen*

Eine Funktion kann mit vollständigem Definitionsbereich spezifiziert werden, wo dann eventuelle Definitionslücken in den Nachbedingungen genauer genannt werden (Beispiel: **tan** auf ganz $\mathbb{R}$).

a1.) Wie hat die Spezifikation mit „vollständigen" Definitionsbereich bei der Funktion **tan** auszusehen?

a2.) Was muß bei Argumenten (die einer Polstelle entsprechen) als Resultat spezifiziert werden (berücksichtigen Sie mindestens zwei Varianten: spezieller Rückgabecode bzw. Ausnahmebedingung)?

a3.) Wie muß ein Aufruf des **tan** in einer umgebenden Programmeinheit aussehen?

Die andere Alternative ist die Benutzung eines eingeschränkten Definitionsbereichs.

b1.) Ein eingeschränkter Definitionsbereich ist etwa durch Konzeption einer geeigneten Vorbedingung realisierbar. Geben Sie eine solche Spezifikation für **tan** an. (*Resultat* ist nur für Elemente des Argumentenbereichs, die die Vorbedingungen erfüllen, definiert.)

b2.) Wie muß ein Aufruf des **tan** in einer umgebenden Programmeinheit aussehen?

Robuste Programme nutzen Vorbedingungen (falls syntaktisch in der aktuellen Programmiersprache unterstützt) oder Ausnahmebedingungen bei der Implementierung einer Funktion wie **tan**. Man kann dann meist Ausnahmebedingungen mit eigenem Namen (etwa: `Definitionslücke bei tan()`) definieren und in der aufrufenden Programmeinheit entweder einen entsprechenden Handler bereitstellen oder auf einen solchen verzichten. Handler werden in umgekehrter Reichenfolge der Aufrufkette gesucht (OS → Hauptprogramm → Unterprogramm ... → **tan(.)**).

c.) Geben Sie in tabellarischer Form Vor- und Nachteile von Variante a und Variante b an. Denken Sie dabei insbesondere an Robustheit, an eventuell wiederholte Abfragen zu Definitionslücken u.ä.

Aufgabe 2. *Spezifikation und Verifikation*

Spezifizieren und verifizieren Sie den folgenden Algorithmus zum Sortieren eines Feldes bezüglich eines Schlüssels:

```
procedure bubblesort;
     var i, j : index;
begin {bubblesort}
     for i := 2 to n do begin
          for j := n downto i do
               if a[j-1].key > a[j].key then begin
                    swap(a[j-1], a[j]);
               end;
     end;
end; {bubblesort}
```

Denken Sie an eine vollständige Spezifikation von `item`, `array[.] of item` und von `index` sowie von `swap()`. Nur falls diese existiert, kann die Verifikation von `bubblesort` durchgeführt werden. (**Denken Sie bei der Invariante an die Splittung in „schon Erreichtes“, „noch Ausstehendes“ und „Gesamtziel“.**)

Aufgabe 3. *Spezifikation und „vollständige Induktion“ bei der Verifikation*

Spezifizieren und verifizieren Sie den folgenden Algorithmus zum Sortieren eines Feldes bezüglich eines Schlüssels:

```
procedure quicksort;
     procedure sort(l, r : index);
          var i, j : index; x : item;
     begin {sort}
          i := l;
          j := r;
          x := a[(l+r) div 2];
          repeat
               while a[i].key < x.key do i := succ(i);
               while x.key < a[j].key do j := pred(j);
               if i <= j then begin
                    swap(a[i], a[j]);
                    i := succ(i);
                    j := pred(j);
               end; {if}
          until i > j;
          if l < j then sort(l,j);
          if i < r then sort(i,r);
     end; {sort}
begin {quicksort}
     sort(1,n);
end; {quicksort}
```

Denken Sie bei der Spezifikation insbesondere an **maxint**, die Definitionsbereiche von `succ` und `pred` (vgl. LIA), die Spezifikation und Verifikation von `sort`, u.ä.

A.6 Übungsblatt 6 — Algorithmenspezifikation in der Numerik

Aufgabe 1. *Spezifikation und Verifikation*

a.) Spezifizieren Sie eine Funktion x^f zur Berechnung (echt) gebrochener Potenzen von x (also: $0 \leq f < 1$).

b.) Schreiben Sie einen entsprechenden Algorithmus. Benutzen Sie dabei nicht den *log()* sondern die Beziehungen:

$$x^f = (\sqrt{x})^{2f}$$

$$x^{1+f} = x \cdot x^f$$

Die Endlichkeit des Algorithmus stellen Sie bitte durch Abbruch nach Erreichen einer gewissen Genauigkeit her. Vergessen Sie dies nicht in der Spezifikation anzugeben!

c.) Schreiben Sie f als Dualzahl und stellen Sie das durch Ihren Algorithmus berechnete Resultat ähnlich wie im Algorithmus zur Berechnung von *ten()* in Abschnitt 1.3.3 (Erinnerung:

$$10^{11} = 10^{1011_B} = \prod_{b_i \neq 0} 10^{2^i} = 10^1 \cdot 10^2 \cdot 10^8$$

$$\uparrow$$

$$1011_B = \begin{array}{lll} 2^0+ & 2^1+ & 2^3 \\ \uparrow & \uparrow & \uparrow \\ i=0 & i=1 & i=3 \end{array}$$

$$10^{\sum_{i=0}^{N} b_i \cdot 2^i} = \prod_{i=0}^{N} 10^{b_i \cdot 2^i} = \prod_{i=0}^{N} (10^{2^i})^{b_i} = \prod_{\substack{i=0 \\ b_i \neq 0}}^{N} 10^{2^i}$$

) dar.

d.) Vergleichen Sie einige numerische Ergebnisse Ihres Algorithmus mit einem gemäß $x^f = \exp(f \cdot \ln(x))$ berechneten Näherungswert. Ist der Definitionsbereich (**floating_overflow**) gleich demjenigen Ihres Algorithmus?

e.) Kombinieren Sie den Algorithmus von Teil b.) mit demjenigen von Aufgabe 3 des Übungsblattes 4 zu einer allgemeinen Potenzfunktion x^y mit:

$$x, y \in \text{IEEE\_real}, y \geq 0$$

(Spezifikation nicht vergessen!)

Aufgabe 2. *Ackermann-Funktion*

Die Ackermann-Funktion wird mathematisch definiert als

$$
\begin{array}{lll}
ackermann : \mathbb{N} \times \mathbb{N} \to \mathbb{N} & & \\
ackermann(n, m) & \underline{\triangle} & \textbf{if } n = 0 \\
& & \textbf{then } m + 1 \\
& & \textbf{else if } m = 0 \\
& & \quad \textbf{then } ackermann(n - 1, 1) \\
& & \quad \textbf{else } ackermann(n - 1, ackermann(n, m - 1))
\end{array}
$$

Spezifizieren Sie eine implementierbare Version. (Berücksichtigen Sie **floating_overflow**.)

Berechnen Sie die Ackermann-Funktion für $(n, m) \in \{(1, 1), (2, 2), (3, 3)\}$.

Das Abfangen von Overflows ist bei der Ackermann-Funktion besonders wichtig, da sie sehr schnell wächst: $ackermann(4, 2)$ besitzt bereits über 21000 Ziffern und $ackermann(4, 4)$ ist größer als $10^{10^{10^{21000}}}$.

Aufgabe 3. *Spezifikation*

Bei numerischen Approximationen wird häufig mit äquidistanten Gittern

$$\Delta x = \frac{b - a}{N}$$

gerechnet. Spezifizieren Sie ein Programm, daß für gegebene $b, a \in$ **IEEE_real** eine absolut äquidistante Einteilung des Intervalls von a nach b in (ungefähr) N Teilintervalle erzeugt und skizzieren Sie einen entsprechenden Algorithmus. (*Hinweis:* Benutzen Sie die LIA-Funktionen, insbesondere $ulp(z)$.)

A.7 Übungsblatt 7 — virtuelle Funktionen und Eigenschaften arithmetischer Primitiva

Aufgabe 1. *SADT*

Verfeinern Sie die Problemlösung `StringToReal` mit Hilfe von SADT-Diagrammen.

Aufgabe 2. *virtuelle Annotations-Funktionen*

Schreiben Sie eine *virtuelle ANNA-Funktion*

$$\texttt{MAX\_IN\_SLICE}(\texttt{X}, \texttt{A}(\texttt{I..J})) \stackrel{\text{def}}{=} \left(X := \max_{j:=I,\ldots,J} A(j) \right)$$

ähnlich wie die Funktion ORDERED:

```
--: function ORDERED(A : VECTOR) return BOOLEAN
--|      where
--|      return for all I, J : A'RANGE => I ≤ J →
--|                              A(I) ≤ A(J);
```

Aufgabe 3. *Eigenschaften von transzendenten Funktionen*

Konvergenzaussagen und Aussagen zur numerischen Genauigkeit des Algorithmus aus Aufgabe 1.b) von Übungsblatt 6 gehen von der Monotonie der Folge

$$\begin{aligned} & x \geq 0.0 \\ & x_0 := x \\ & x_i := \sqrt{x_{i-1}} \quad \forall\ i \in \{1, 2, \ldots\} \end{aligned}$$

aus.

Welche Forderungen sollte man an die transzendente Funktion `sqrt` stellen, damit eine solche Argumentation bei der Verifikation von „echten“ Algorithmen ebenfalls greifen kann?

Spezifizieren Sie `sqrt` vollständig.

Aufgabe 4. *Work around*

Bei Konversionsaufgaben mußten Sie häufig eine genaue Operation

$$x \mapsto 10.0 \cdot x,\ \ x \in \texttt{IEEE\_real}$$

als Vorbedingung voraussetzen, um das gewünschte Ziel erreichen zu können.

Nehmen wir an, daß Sie in einer höheren Programmiersprache eine Dualarithmetik nutzen, die „genau“ mit Zweierpotenzen multiplizieren kann (Operation in der LIA-Arithmetik: `scale`), daß aber durch einen Bug die Multiplikation mit 10.0 nicht exakt ausgeführt wird.

Schreiben Sie eine Funktion `ten`, die mit Hilfe von `scale` die Funktion

$$x \mapsto 10.0 \cdot x$$

möglichst genau berechnet[1], spezifizieren Sie diese Funktion und verifizieren Sie!

[1] Hinweis: $\cdot 10.0 = \cdot 2^3 \cdot 1.25$

Index

Literaturverzeichnis

[1] *The Programming Language Ada, ANSI/MIL-STD-1815A-1983*, number 155 in Lecture Notes in Computer Science, Berlin, 1983. Springer-Verlag.

[2] *ANSI C, ANSI-STD X3.159-1989*. American National Standard Institute, X3J11, 1989.

[3] *Third Working Draft Modula-2 Standard, Document D106/N336*. British Standards Institute, Working Group IST/5/13, 1989.

[4] *Draft international standard ISO/IEC CD 10967-1:1992, Information Technology, Part 1: Language Independent Arithmetic, Part 1: Integer and Floating Point Arithmetic*. International Standardization Organization (ISO), 1992.

[5] *RISCS-FORUM Digest*. FORUM ON RISKS TO THE PUBLIC IN COMPUTERS AND RELATED SYSTEMS, ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator, moderated Internet discussion group, since 1980.

[6] Suad Alagić und Michael A. Arbib. *The Design of Well Structured and Correct Programms*. Springer, New York, 1978.

[7] Nancy Baxter, Ed Dubinsky und Gary Levin. *Learning Discrete Mathematics with ISETL*. Springer, New York, 1989.

[8] Valdis Andris Bērzins und Luqi. *Software Engineering with Abstractions*. Addison Wesley, Reading, 1991.

[9] W. F. Clocksin und C. S. Mellish. *Programming in Prolog*. Springer, New York, 1981.

[10] Edward Cohen. *Programming in the 1990s. An Introduction to the Calculation of Programms*. Springer, New York, 1990.

[11] O. J. Dahl, E. W. Dijkstra und C. A. Hoare. *Structured Programming*. Academic Press, London, 1972.

[12] Department of Defense Computer Security Center. *Orange Book, Department of Defense Trusted Computer System Evaluation Criteria DOD 5200.28-STD*, Fort George G. Meade, 1985.

[13] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.

[14] Edsger W. Dijkstra und W. H. J. Feijen. *Methodik des Programmierens.* Addison Wesley Verlag (Deutschland), 1985.

[15] Edsger W. Dijkstra und Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer, New York, 1990.

[16] Margaret A. Ellis und Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, 1990.

[17] Elfriede Fehr. *Semantik von Programmiersprachen.* Springer, Berlin, 1989.

[18] Davis Gries. *The Science of Programming.* Springer, New York, 1981.

[19] IEEE, Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[20] IFIP Working Group 2.5 (Mathematical Software). *Exception Handling in FORTRAN 90*, October 1992.

[21] *DUDEN Informatik, Ein Sachlexikon für Studium und Praxis.* Bibliographisches Institut, Mannheim, 1988.

[22] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall, New York, second edition, 1990.

[23] Cliff B. Jones und R. C. Shaw. *Case Studies in Systematic Software Development.* Prentice Hall, New York, 1990.

[24] Brian W. Kernighan und P. J. Plauger. *The Elements of Programming Style.* McGraw-Hill, New York, second edition, 1978.

[25] Herbert Klaeren. *Vom Problem zum Programm. Eine Einführung in die Informatik.* Teubner, Stuttgart, 1991.

[26] Ulrich W. Kulisch und Willard L. Miranker. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[27] David Luckham. *Programming with Specifications; An Introduction to ANNA, a language for specifying ADA Programms.* Springer, New York, 1990.

[28] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner und Olaf Owe. *ANNA, a Language for Annotating Ada Programms, Reference Manual.* Springer, Berlin, 1987.

[29] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[30] Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, New York, 1990.

[31] Bertrand Meyer. *Eiffel: The Libraries, TR-EI-7ILI Version 2.3.* Interactive Software Engineering Inc., 1991.

[32] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, New York, 1992.

[33] Fred Ris, Ed Barkmeyer, Craig Schaffert und Peter Farkas. When floating-point addition isn't commutative. *SIGNUM Newsletter*, 28 number 1:8–13, January 1993. Association for Computing Machinery.

[34] D. Russel und G. T. Gangemi Sr. *Computer Security Basics.* O'Reilly and Associates, Sebastopol, 1991.

[35] Arno Schulz. *Software-Entwurf, Methoden und Werkzeuge.* R. Oldenbourg Verlag, München, 1988.

[36] David E. Smith. *X Application Evolution using Wcl2.4 (Widget Creation Library).* Technical report, Siemens-Nixdorf Informationssysteme AG, München, 1993. X Technical Conferenence Tutorial, Boston.

[37] Sun Soft, Sun Microsystems. *Open Windows Developer's Guide 3.0.1 User's Guide*, Mountain View, 1992.

[38] Niklaus Wirth. *Algorithmen und Datenstrukturen.* Teubner, Stuttgart, 1979.

[39] Mario Wolczko. *Typesetting BSI VDM with* LaTeX. Manchester, 1992.

[40] Zentralstelle für Sicherheit in der Informationstechnik ZSI, editor. *IT-Sicherheitskriterien.* Bundesanzeiger Verlagsgesellschaft, 1989.