

Im folgenden finden Sie teils vollständige Lösungsvorschläge, zum Teil aber auch nur Hinweise für die Lösung der Aufgaben zu Abschnitt 4.3. Sie haben damit die Chance, noch ein wenig nachzudenken und kreativ zu sein.

Aufgabe 4.20

Feststellung: sowohl bei ASCII als auch bei EBCDIC liegt zwischen jedem Klein- und dem entsprechenden Großbuchstaben eine feste Distanz d . Bezeichnet $\text{int}(x)$ eine Funktion zur Berechnung des Kodewertes des Zeichens x , so gilt:

$$d_{ASCII} = |\text{int}('A') - \text{int}('a')| = |65 - 97| = 32 = 0010\ 0000 = 20_{16}$$

$$d_{EBCDIC} = |\text{int}('A') - \text{int}('a')| = |193 - 129| = 64 = 0100\ 0000 = 40_{16}$$

Somit gilt für die Umwandlung von Klein- in Großbuchstaben in

ASCII: Subtrahiere 32 vom Kodewert des Kleinbuchstabens (dezimal) bzw. blende das Bit mit der Wertigkeit $2^5 = 32$ aus durch bitweises UND (&) mit $1101\ 1111 = -33$ oder $0101\ 1111 = 95$.

Beispiel: $\text{int}('A') = \text{int}('a') \& (-33)$ entspricht $0100\ 0001 = 0110\ 0001 \& 1101\ 1111$.

EBCDIC: Addiere 64 zum Kodewert des Kleinbuchstabens (dezimal) bzw. füge ein Bit mit der Wertigkeit $2^6 = 64$ ein (bitweises ODER (|)) mit $0100\ 0000 = 64$.

Beispiel: $\text{int}('A') = \text{int}('a') | 64$ entspricht $1100\ 0001 = 1000\ 0001 | 0100\ 0000$.

Vergleichen Sie hierzu auch das Beispiel 4.29

Aufgabe 4.21

Lösung in Arbeit.

Aufgabe 4.22

Siehe Abruf: Gleitkommadarstellung nach IEEE-757

Aufgabe 4.23

Konstante Daten $\text{maxlaenge} = 4 \in \text{Integer};$ // Legt maximale Stellenzahl von n fest
...

Block Eingabe (**aus**: $n \in \text{Integer}$)

Daten *Zahl* \in **Feld Char** [$1..\text{maxlaenge}$];
 i \in **Integer**;
 Zeilenende, Kontrolle \in **Boole**;

{ **Für** $i = 0$ **solange** $i \leq \text{maxlaenge}$ **mit** $i = i + 1$ **führe aus**
 Zahl[i] = '' ; // Initialisierung mit Leerzeichen

Wiederhole

Bildschirm_löschen();
 $i := 0$; $n = 0$; *Zeilenende* = f ;
 Bildschirmausgabe: "Eingabe: n = ";

```

Solange Zeilenende = f führe aus
{  i := i+1;
  Falls i ≤ maxlaenge
  dann Lese_Taste( aus: Zahl[i]);
  Falls '0' ≤ Zahl[i] ≤ '9'
  dann { Kontrolle = w;
          n := n · 10 + ord(Zahl[i]) - ord('0'); // Berechnung der Zahl aus den eingelese-
                                                // nen Ziffern nach dem Horner Schema
        }
  sonst Falls Zahl[i] = 'CR'
  dann { Zeilenende = w;
          Kontrolle = w;
        }
  sonst { Kontrolle = f;
          Bildschirmausgabe: "Falsche Eingabe! Weiter mit <Return>";
          Zeilenende = w;
          Für i = 0 solange i ≤ maxlaenge mit i = i + 1 führe aus
            Zahl[i] := '';
          }
  }
solange Kontrolle = f;
}

```

Der parameterlose Block *Bildschirm_löschen* löscht beim Aufruf den Bildschirm. Ein Unterprogramm dieser Art steht in allen gängigen Entwicklungsumgebungen als Bibliotheksfunktion zur Verfügung. Nachfolgend noch eine Erläuterung zur Berechnung des Wertes einer Dezimalzahl aus ihren Ziffern.

Seien $z_2 = 5$, $z_1 = 3$ und $z_0 = 1$ die Ziffern der Dezimalzahl n . Dann gilt

$$n = z_2 \cdot 10^2 + z_1 \cdot 10^1 + z_0 \cdot 10^0 = (z_2 \cdot 10 + z_1) \cdot 10 + z_0 = (5 \cdot 10 + 3) \cdot 10 + 1 = 531$$

Stellenwertverfahren

HORNER-Darstellung

Aufgabe 4.24

Im ASCII-Zeichensatz liegen die Codewerte der Großbuchstaben im Bereich 65 bis 90 und die Kleinbuchstaben im Bereich 97 bis 122. Eine Umwandlung kann also durch Erhöhung des Kodewertes um 32 erfolgen.

Block *Upper_to_Lower* (**mit:** *Buchstabe* ∈ **Char**)

```

{ Falls 65 ≤ ord(Buchstabe) ≤ 90 // Buchstabe repräsentiert Großbuchstaben
  dann Buchstabe := chr(ord(Buchstabe) + 32);
}

```

Aufgabe 4.25

...
Konstante Daten *maxlaenge* = 31 ∈ Integer; // Legt maximale Stellenzahl der hexadezi-
// malen Zahlen *hex1* und *hex2* auf 30 fest.

Typdefinition *hexzahl* = Feld Char [1..*maxlaenge*]
...

Block *Laenge* (**ein:** *hex* ∈ *hexzahl* **aus:** *l* ∈ Integer)
?? Ermittelt die Anzahl der Ziffern der hexadezimalen Zahl *hex*. ??

Block *Lower_to_Upper* (**mit:** *Buchstabe* ∈ Char)
?? Wandelt *Buchstabe* in Großbuchstaben ??

Block *Add_Hex* (**ein:** *hex1*, *hex2* ∈ *hexzahl*, **aus:** *erg_hex* ∈ *hexzahl*)
// *hex1* und *hex2* enthalten die zu addierenden Hexzahlen, *erg_hex* das Ergebnis der Addition.

```
Daten      d2h      ∈ Feld Char [0..15];
                                     // Konversionsfeld Dezimal nach Hex
                                     // Index fuer d2h
                                     // Index fuer hex1
                                     // Index fuer hex2
                                     // Index fuer erg_hex
                                     // Ergebnis einer stellenweisen Addition
                                     // Uebertrag
                                     ∈ Integer;

{
  ue := 0; l3 := 31
  Für i = 0 solange i ≤ 9 mit i = i + 1 führe aus // Initialisierung des Konversionsfeldes:
    d2h[i] := chr(48 + i); // Ziffern 0 bis 9
  Für i = 10 solange i ≤ 15 mit i = i + 1 führe aus
    d2h[i] := chr(55 + i); // Ziffern A bis F
  Laenge ( ein: hex1; aus: l1 ∈ Integer ); // Ermittlung der Laenge von hex1
  Laenge ( ein: hex2; aus: l2 ∈ Integer ); // Ermittlung der Laenge von hex2

  Solange (l1 ≥ 0) ∨ (l2 ≥ 0) führe aus
  {
    w := ue;
    Für i = 0 solange i ≤ 15 mit i = i + 1 führe aus
    {
      Falls hex1[l1] = d2h[i] // Dezimalwert fuer Hexziffer gefunden
      dann w := w + i;
      Falls hex2[l2] = d2h[i]
      dann w := w + i; // Summe der Hexziffern dezimal gebildet
    }
    Falls w > 15 // Übertrag
    dann { ue := 1;
           w := w - 16;
          }
    sonst ue := 0;
    erg_hex[l3] := d2h[w];
    l1 := l1 - 1;
    l2 := l2 - 1;
    l3 := l3 - 1;
  }
}
```

Aufgabe 4.26

- a) Für das Skalarprodukt zweier Vektoren $\mathbf{a} = (a_1, a_2, a_3)$ und $\mathbf{b} = (b_1, b_2, b_3)$ in einer Orthonormalbasis des \mathbf{R}^3 gilt: $\mathbf{a} \bullet \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$. Damit

...
Typdefinition *Vektor* = **Feld Real** [1..3]
 ...

Block *Skal_Prod* (**ein:** $a, b \in \text{Vektor}$ **aus:** $x \in \text{Real}$)
Daten $i \in \text{Integer};$

```
{
  x := 0;
  Für i = 1 solange i ≤ 3 mit i = i + 1 führe aus
    x := x + a[i] · b[i];
}
```

- b) Für das Vektorprodukt

Für den Betrag des Vektorprodukts zweier Vektoren $\mathbf{a} = (a_1, a_2, a_3)$ und $\mathbf{b} = (b_1, b_2, b_3)$ in einer Orthonormalbasis des \mathbf{R}^3 , welche ein Rechtssystem bildet, gilt:

$$|\mathbf{a} \times \mathbf{b}| = \sqrt{(a_2 b_3 - a_3 b_2)^2 + (a_3 b_1 - a_1 b_3)^2 + (a_1 b_2 - a_2 b_1)^2}.$$

Damit

...
Typdefinition *Vektor* = **Feld Real** [1..3]
 ...

Block *Abs_Vek_Prod* (**ein:** $a, b \in \text{Vektor}$ **aus:** $abs_v \in \text{Real}$)

```
{
  abs_v := sqrt((a[2]·b[3] - a[3]·b[2]) · (a[2]·b[3] - a[3]·b[2])
    + (a[3]·b[1] - a[1]·b[3]) · (a[3]·b[1] - a[1]·b[3])
    + (a[1]·b[2] - a[2]·b[1]) · (a[1]·b[2] - a[2]·b[1]));
}
```

Aufgabe 4.27

- a) Die Addition zweier Matrizen ist als Addition der entsprechenden Elemente definiert:

$$\mathbf{S} = \mathbf{A} + \mathbf{B} = (s_{ij})_{mm} \quad \text{mit} \quad s_{ij} = a_{ij} + b_{ij}.$$

Diese Vorschrift wird durch den folgenden Algorithmus realisiert:

...
Konstante Daten $mmax = 5 \in \text{Integer};$ // Legt die maximale Zahl der Zeilen und
 // Spalten der Matrizen auf (z.B.) 5 fest.

Typdefinition *Matrix* = **Feld Real** [1..mmax, 1..mmax]
 ...

Block *Matrix_Sum* (**ein:** $A, B \in \text{Matrix}$ **aus:** $S \in \text{Matrix}$)

Daten $i, j \in \text{Integer};$

```
{
  Für i = 1 solange i ≤ mmax mit i = i + 1 führe aus
    Für j = 1 solange j ≤ mmax mit j = j + 1 führe aus
      s[i, j] := a[i, j] + b[i, j];
}
```

b) Die Multiplikation zweier Matrizen ist durch die folgende Vorschrift definiert:

$$\mathbf{P} = \mathbf{A} \cdot \mathbf{B} = (p_{ij})_{mm} \quad \text{mit} \quad p_{ij} = \sum_{r=1}^{mmax} a_{ir} + b_{rj} \quad .$$

Diese Vorschrift wird durch den folgenden Algorithmus realisiert:

```

...
Konstante Daten mmax = 5 ∈ Integer;           // Legt die maximale Zahl der Zeilen und
                                                    // Spalten der Matrizen auf (z.B.) 5 fest.
Typdefinition Matrix = Feld Real [1..mmax, 1..mmax]
...

Block Matrix_Prod ( ein: A, B ∈ Matrix; aus: P ∈ Matrix )
  Daten   i, j, r ∈ Integer;

  {
    Für i = 1 solange i ≤ mmax mit i = i + 1 führe aus
      Für j = 1 solange j ≤ mmax mit j = j + 1 führe aus
        { p[i, j] := 0;
          Für r = 1 solange r ≤ mmax mit r = j + 1 führe aus
            p[i, j] := p[i, j] + a[i, r] · b[r, j];
          }
        }
  }

```

Aufgabe 4.28

Es wird eine 3 × 4 - Matrix angenommen.

```

...
Konstante Daten mmax = 3 ∈ Integer;           // Größter Zeilenindex
                  nmax = 4 ∈ Integer           // Größter Spaltenindex

Typdefinition Matrix = Feld Real [1..mmax, 1..nmax]
...

Block Max_ij ( mit: A ∈ Matrix; aus: imax, jmax ∈ Integer )
// Ermittelt die Indices imax und jmax des Pivotelementes
  Daten   i, j, imax, jmax ∈ Integer;

  {
    imax := 1; jmax := 1;
    Für i = 1 solange i ≤ mmax mit i = i + 1 führe aus
      Für j = 1 solange j ≤ nmax mit j = j + 1 führe aus
        Falls fabs(A[i, j]) > fabs(A[imax, jmax])
          dann { imax := i;
                  jmax := j;
                }
      }
  }

```

```

Block Zeilentausch ( ein: i, j ∈ Integer; mit: A ∈ Matrix )
// Vertauscht in der Matrix A die Zeilen i und j
  Daten   k, h ∈ Integer;

  {
    Für k = 1 solange k ≤ nmax mit k = k + 1 führe aus
      { h := A[i, k];
        A[i, k] := A[j, k];
      }
  }

```

```

    A[j, k] := help;
  }
}

```

Block Spaltentausch (**ein:** $i, j \in \text{Integer}$; **mit:** $A \in \text{Matrix}$)

// Vertauscht in der Matrix A die Spalten i und j

Daten $k, h \in \text{Integer}$;

```

{
  Für  $k = 1$  solange  $k \leq mmax$  mit  $k = k + 1$  führe aus
  {
     $h := A[k, i]$ ;
     $A[k, i] := A[k, j]$ ;
     $A[k, j] := h$ ;
  }
}

```

Block Pivot (**mit:** $A \in \text{Matrix}$)

// Vertauscht zwei Zeilen und zwei Spalten in der Matrix A so, daß das Pivotelement den Platz $A[1, 1]$ // der Matrix einnimmt

Daten $imax, jmax \in \text{Integer}$;

```

{
  Max_ij ( mit:  $A \in \text{Matrix}$ ; aus:  $imax, jmax \in \text{Integer}$  )
  Zeilentausch ( ein: 1,  $imax$ ; mit: A )
  Spaltentausch ( ein: 1,  $jmax$ ; mit: A )
}

```

Aufgabe 4.29

Für die in den Punkten a) ... d) angegeben Blöcke seien die folgende Typdefinition vereinbart:

Typdefinition $\text{Listenzeiger} = \text{Addr Segment}$;
 $\text{Segment} = \text{Verbund}$

```

{
  Wert  $\in \text{String}$ 
  Naechstes  $\in \text{Listenzeiger}$ 
};

```

Anfang und Ende sind Zeiger vom Typ *Listenzeiger* und zeigen auf das erste bzw. letzte Element der Liste.

a) **Block Einfügen** (**ein:** $\text{Wert} \in \text{String}$; **mit:** $\text{Vorgaenger} \in \text{Segment}$)

// Fügt ein neues Segment nach "Vorgaenger" ein und initialisiert die Komponente Wert.

Daten $\text{Neues} \in \text{Listenzeiger}$;

```

{
  Neues := Erzeuge Segment;
  deref Neues.Wert := Wert;
  deref Neues.Naechstes := Vorgaenger.Naechstes;
  Vorgaenger.Naechstes := Neues;
}

```

b) **Block Löschen** (**mit:** $\text{Vorgaenger} \in \text{Segment}$; **aus:** $\text{Wert} \in \text{String}$)

// Löscht das Element nach "Vorgänger". Der Inhalt wird unter "Wert" zurückgegeben.

Daten $\text{Altes} \in \text{Listenzeiger}$;

```

{
  Altes := Vorgaenger.Naechstes;
  Vorgaenger.Naechstes := deref Altes.Naechstes;
}

```

```

    Wert := deref Altes.Wert;
    Loesche(Altes);
}

```

c) **Block Leer** (**ein:** Anfang \in Listenzeiger **aus:** leer \in **Boole**)
 // Liefert "wahr", wenn die Liste leer ist.

```

{   leer := (Anfang = Nil);
}

```

d) **Block Initialisiere** (**mit:** Anfang \in Listenzeiger)
 // Initialisiert die Liste mit Anfang := Nil

```

{   Anfang := Nil;
}

```

Die Erweiterung dieser Lösungen auf doppelt verkettete Listen bleibe dem interessierten Leser überlassen.

Aufgabe 4.30

Zur Realisierung der einfach verketteten Liste werden zwei "parallele" Felder *Segment* und *Naechstes* verwendet. *Segment* enthält die Daten (hier vom Typ *String*). *Naechstes* ist das Verkettungsfeld und enthält die Verweise (Indices) auf die Elemente von *Segment*. Nachfolgend zunächst das Prinzip:

```

...
Konstante Daten   mmax = 100   $\in$  Integer;           // maximale Laenge der Liste

Typdefinition     Datenfeld   = Feld String [0..mmax];
                    Indexfeld   = Feld Integer [0..mmax];

Daten             Segment      $\in$  Datenfeld;           // Datenfeld
                    Naechstes    $\in$  Indexfeld;           // Verkettungsfeld
                    x,           // zeigt auf den naechsten freien Speicher
                    Kopf,        // zeigt auf den Anfang der Liste
                    p             $\in$  Integer;           // Verkettungsindex
...

```

Block Einfügen (**ein:** Vorgaenger \in **Integer**, Wert \in **String**; **aus:** Fehler_Code \in **Integer**)
 // Fügt ein neues Segment nach "Vorgaenger" ein und initialisiert es mit Wert

```

    verwendet Segment, Naechstes, x;
{   x := x + 1;           // "Erzeugt" ein neues Listenelement
    Falls x > mmax
    dann Fehler_Code := ? ... ?; // kein Platz für neues Segment
    sonst { Segment[x] := Wert;
           Naechstes[x] := Naechstes[Vorgaenger];
           Naechstes[Vorgaenger] := x;
         }
}

```

Block Löschen (**ein:** Vorgaenger \in **Integer**; **aus:** Wert \in **String**)
 // Löscht das Element nach "Vorgänger". Der Inhalt wird unter "Wert" zurückgegeben.

```

    verwendet Segment, Naechstes;

```

```

{   Wert := Segment[Naechstes[Vorgaenger]];
    Naechstes[Vorgaenger] := Naechstes[Naechstes[Vorgaenger]];
}

```

Block Initialisiere ()

// Initialisiert die Liste

```

verwendet Naechstes, x, Kopf, p;
{   Kopf := 0;
    p := 1;
    x := 1;
    Naechstes[Kopf] := p;
    Naechstes[p] := p;
}

```

Es bleibt im Wesentlichen noch das Problem zu lösen, die durch Aufrufe von *Löschen* im Feld *Segment* frei werdenden Elemente für die Wiederverwendung verfügbar zu machen. Dieses Problem der "Speicherplatzverwaltung" kann durch Einführen einer zweiten Liste, der *free-list*, gelöst werden. In die *free-list* werden durch Aufruf von *Einfügen* freie Elemente eingetragen und durch Aufruf von *Löschen* freie Elemente ausgetragen. Die *free-list* kann innerhalb der Felder *Segment* und *Naechstes* implementiert werden. Hierzu ist ein zweiter Kopf zu verwalten. Die Ausarbeitung dieses Lösungsvorschlags sei dem Leser als Übung überlassen.

Aufgabe 4.31

Die folgende Definition eines ADT "FIFO-Warteschlange für ganze Zahlen" lehnt sich an J. GUTTAG: "Abstract data types and the development of data structures", *Comm. ACM*, 20(6), 396-405 (1977) an.

ADT *FIFO_Warteschlange*; // kurz: *FIFO*

Import

Typen **Integer, Boole;**
Konstante Daten *FIFO_Groesse* > 0;

Export

Typdefinition *Fehlercode* = (*OK*, *FIFO_Ueberlauf*, *FIFO_Unterlauf*, *FIFO_Leer*);

erzeuge:	$\phi \rightarrow FIFO$	// erzeugt eine leere Warteschlange
bringen:	$FIFO \times \mathbf{Integer} \rightarrow FIFO (\times Fehler_Code)$	// fügt ein neues Element am Ende an
holen:	$FIFO \rightarrow FIFO (\times Fehler_Code)$	// entfernt das Element am Anfang
kopf:	$FIFO \rightarrow \mathbf{Integer} (\times Fehler_Code)$	// Abfrage des Elementes am Anfang
ist_leer:	$FIFO \rightarrow \mathbf{Boole}$	// Abfrage ob Warteschlange leer
ist_voll:	$FIFO \rightarrow \mathbf{Boole}$	// Abfrage ob Warteschlange voll
laenge:	$FIFO \rightarrow \mathbf{Integer}$	// liefert die Länge der Warteschlange
loesche:	$FIFO \rightarrow \phi$	// löscht eine Warteschlange

Axiome sei $x \in FIFO$ und $i \in \mathbf{Integer}$

LEER:	$laenge(x) = 0 \Leftrightarrow ist_leer(x) = wahr$
VOLL:	$laenge(x) = FIFO_Groesse \Leftrightarrow ist_voll(x) = wahr$
ANZAHL:	$ist_voll(x) = falsch \Rightarrow laenge(bringen(x, i)) = laenge(x) + 1$
ANZAHL_0:	$laenge(erzeuge) = 0$
KOPF:	$kopf(bringen(x, i)) = \mathbf{Falls} \text{ leer}(x)$

```

                                dann  i
                                sonst kopf(x)
HOLEN:    holen(bringen(x, i)) = Falls leer(x)
                                dann  erzeuge
                                sonst  bringen(holen(x), i)

OK1:      ist_voll(x) = falsch  $\Rightarrow$  bringen(x, i) = Fehler_Code: OK
OK2:      ist_leer(x) = falsch  $\Rightarrow$  holen(x, i) = Fehler_Code: OK
OK3:      ist_leer(x) = falsch  $\Rightarrow$  kopf(x, i) = Fehler_Code: OK

F1:       kopf(erzeuge) = Fehler_Code: FIFO_Leer
F2:       holen(erzeuge) = Fehler_Code: FIFO_Unterlauf
F3:       ist_voll(x) = wahr  $\Rightarrow$  bringen(x, i) = Fehler_Code: FIFO_Ueberlauf

```

Aufgabe 4.32

Es wird der Einfachheit halber ein ADT für eine Menge mit Elementen aus **Char** definiert. An Stelle einer axiomatischen Spezifikation der Operationen (siehe hierzu etwa: Ian Sommerville: „Software Engineering“, Addison Wesley (Deutschland) GmbH, Bonn (1987)) werden hier Vor- und Nachbedingungen verwendet (*operationelle Spezifikation*). Dies ist für den algebraisch wenig Geübten eine einfachere und für praktische Anwendungen angemessenere Methode.

```
ADT Zeichen_Menge;           // kurz: Menge
```

Import

```

Typen           Char, Boole, Integer;
Konstante Daten Max_Umfang > 0;

```

Export

```
Typdefinition Fehlercode = (OK, Menge_Leer, Mengen_Ueberlauf);
```

```

erzeuge:   $\phi \rightarrow Menge$            // erzeugt eine leere Menge
add:       $Menge \times Char \rightarrow Menge (\times Fehler\_Code)$  // fügt ein Element hinzu
sub:       $Menge \times Char \rightarrow Menge (\times Fehler\_Code)$  // entfernt ein Element
ist_leer:  $Menge \rightarrow Boole$          // Abfrage ob Menge leer
ist_in:    $Menge \times Char \rightarrow Boole$  // Abfrage ob Element aus der Menge ist
umfang:    $Menge \rightarrow Integer$        // liefert die Mächtigkeit einer Menge
loesche:   $Menge \rightarrow \phi$            // entfernt eine Menge

```

Operationen

```

BRINGEN:  vor   $S \in Menge, c \in Char$ 
           nach  $add(S, c) = S \cup \{c\}$ 

HOLEN:    vor   $S \in Menge, e \in S$ 
           nach  $sub(S, c) = S \setminus \{c\}$ 

LEER:     vor   $S \in Menge$ 
           nach Falls  $S = \phi$ 
                dann  $ist\_leer(S) = wahr$ 
                sonst  $ist\_leer(S) = falsch$ 

```

IN: **vor** $S \in Menge, c \in Char$
 nach Falls $c \in Menge$
 dann $ist_in(S, c) = wahr;$
 sonst $ist_in(S, c) = falsch;$

UMFANG: **vor** $S \in Menge, S =$ enthält m Elemente (Mächtigkeit von S)
 nach $umfang(S) = m$

OK1: **vor** $S \in Menge, umfang(S) < Max_Umfang$
 nach $add(S, c) = Fehler_Code: OK$

OK2: **vor** $S \in Menge, ist_leer(S) = falsch$
 nach $sub(S, e) = Fehler_Code: OK$

F1: **vor** $S \in Menge, e \in S$
 nach $sub(erzeuge, e) = Fehler_Code: Menge_Leer$

F2: **vor** $S \in Menge, c \in Char, umfang(S) = Max_Umfang$
 nach $add(S, c) = Fehler_Code: Mengen_Ueberlauf$

Aufgabe 4.33

Die Lösung ist im Stile von Aufg. 4.32.

ADT *Komplexe_Zahl*;

Import

Typen **Real**;
 Typdefinition $C = Verbund$
 { $Re \in Real$;
 $Im \in Real$;
 }

Export

erzeuge: $\phi \rightarrow C$ // erzeugt eine komplexe Variable
 init: $C \times Real \times Real \rightarrow C$ // Initialisierung
 := $C \times C \rightarrow C$ // Zuweisung
 + $C \times C \rightarrow C$ // Addition
 * $C \times C \rightarrow C$ // Multiplikation
 abs: $C \rightarrow Real$ // Betragsfunktion
 re: $C \rightarrow Real$ // Realteil lesen
 im: $C \rightarrow Real$ // Imagimärteil lesen
 loesche: $C \rightarrow \phi$ // entfernt eine komplexe Variable

Operationen

REAL: **vor** $z \in C$
 nach $re(z) = z.Re$

IMAG: **vor** $z \in C$
 nach $im(z) = z.Im$

