Algorithmen und Datenstrukturen (Informatik III)

Prof. Dr. Hans-Jürgen Buhl

1999

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

Interner Bericht der Integrierten Arbeitsgruppe
Mathematische Probleme aus dem Ingenieurbereich
IAGMPI – 9702
November 1999

Praktische Informatik 04

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Algorithmen

1.1 Ein intuitiver Algorithmenbegriff

Definition 1.1.1
Ein *Algorithmus* ist eine endliche Folge von eindeutigen Anweisungen, mittels derer in endlich vielen Schritten aus vorgegebenen spezifizierten Eingabegrößen spezifizierte Ergebnisgrößen gewonnen werden.
Ein Algorithmus berechnet also eine Funktion der Eingabe.
Dabei müssen folgende Punkte erfüllt sein:

1. Ein- und Ausgabe sind genau festgelegt; zu jeder Sorte von Eingabegrößen gibt es (genau) eine gültige Ausgabe

   Frage : „Welche Eingabegrößen sind erlaubt und/oder sinnvoll ?“

   Frage : „Welche Funktion soll der Algorithmus berechnen ?“
   (→ Problemspezifikation)

   Frage : „Berechnet der Algorithmus wirklich die spezifizierte Funktion ?“
   (→ Korrektheitsuntersuchung)

2. Jede Anweisung darf nur endlich viele Schritte benötigen und nur endlich oft ausgeführt werden.

   Frage : „Wie sehen die Schritte aus ?“
   (→ Maschinenmodell)

   Frage : „Terminiert der Algorithmus, d.h. liefert er nach endlich vielen Schritten eine Ausgabe ?“
   (→ Terminierung)

3. Jede Anweisung muß ein eindeutig (reproduzierbares) Resultat haben (→ Definitheit)

4. Das Aufschreiben des Algorithmus darf nur endlich viel Platz beanspruchen (also nicht: „usw.“)

5. Möglichst geringer Ressourcenverbrauch wie Speicher, Rechenzeit (→ Effizienz)

6. Der Algorithmus beruht auf einer nachvollziehbaren Idee und ist verständlich formuliert. (→ Verständlichkeit)

Beispiel: (Euklidischer Algorithmus)
Geg.: Zwei Zahlen $m, n \in \mathbb{N}$, $m > n$
Ges.: Der größte gemeinsame Teiler $ggT(m, n)$

1. [Division mit Rest]
   Berechne $m = n \cdot q + r$, $r, q \in \mathbb{N}_0$, $0 \leq r < n$

2. [Ergebnis]
   Falls $r = 0$ beende Algorithmus, $ggT = n$

3. [Ersetzen]
   $m \leftarrow n$, $n \leftarrow r$, gehe zu 1.

(”$\leftarrow$” heißt: „wird ersetzt durch“)

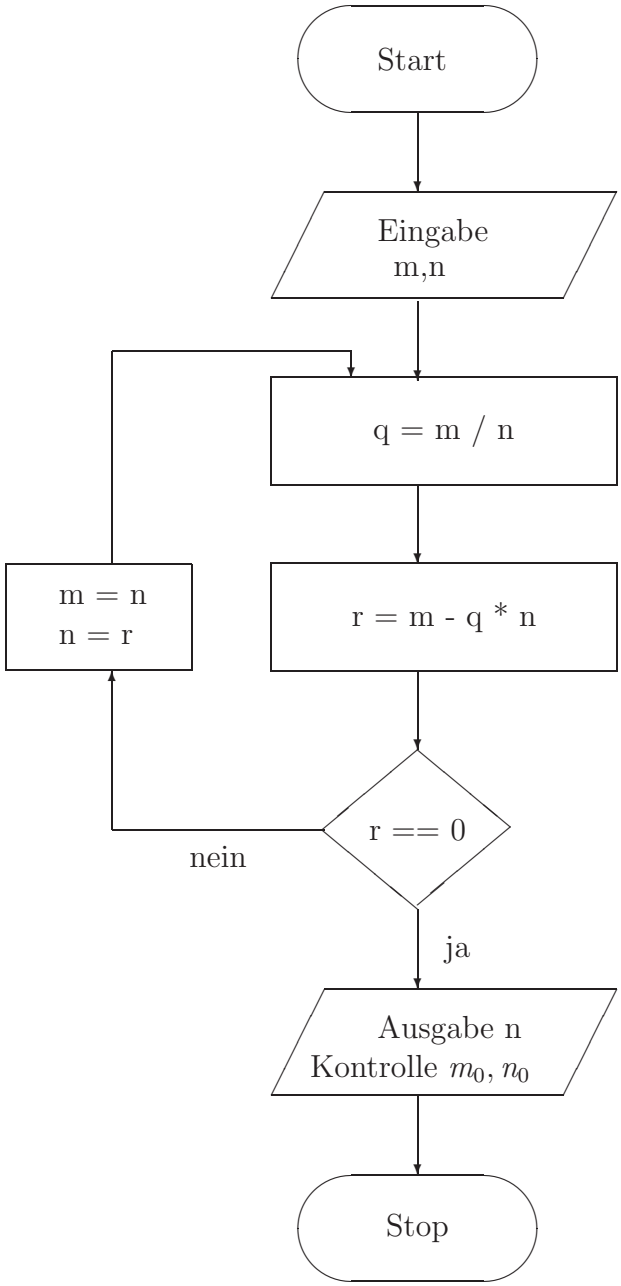
Zahlenbeispiel: $m = 2754$, $n = 378$

Tabelle 1.1: Zahlenbeispiel

Status nach	Schritt-Nr.	m	n	q	r	ggT
Startwerte	-	2754	378	?	?	?
1. Durchlauf	1.	2754	378	7	108	?
	3.	378	108	7	108	?
2. Durchlauf	1.	378	108	3	54	?
	3.	108	54	3	54	?
3. Durchlauf	1.	108	54	2	0	?
	2.	108	54	2	0	$\boxed{54}$

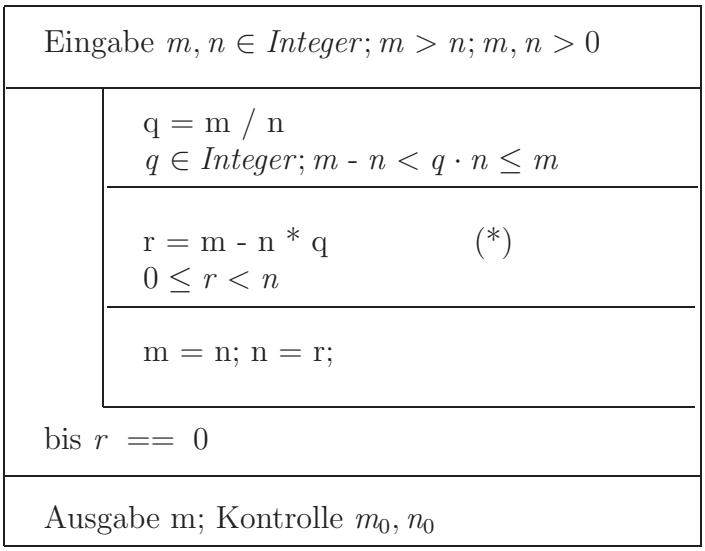
Hier nun das zugehörige *Flußdiagramm*:

Abbildung 1.1: Flußdiagramm



oder besser noch ein Struktogramm:

Abbildung 1.2: Struktogramm



- *Spezifikation* hier:
  Eingabe : $m, n \in \mathbb{N}$, $m > n$
  Ausgabe : $ggT = ggT(m, n) \in \mathbb{N}$, d.h. diejenige natürliche Zahl, die n und m teilt und für die gilt : Teilt $d \in \mathbb{N}$ sowohl n als auch m, so ist d Teiler von ggT.

- *Korrektheit:* Ist ggT aus dem Algorithmus wirklich der ggT aus der Spezifikation? Zu zeigen:

  - teilt d die Eingabegrößen n und m, so teilt d auch jeweils die n, m, die in Schritt 3 erzeugt werden
    ($\rightarrow$ d teilt ggT).
  - ggT aus Schritt 2 ist gemeinsamer Teiler aller n, m, die im Algorithmus vorkommen

  Oder einfacher die Invariante $ggt(m, n) = ggt(n, r)$ in (*) von Abb. 1.2

- *Terminierung:* Zu zeigen: $r = 0$ wird nach endlich vielen Durchläufen erreicht. Ist r_i der Wert von r in Schritt 2 im i-ten Durchlauf, so gilt wegen Schritt 1 und 3

$$r_i < r_{i\text{-}1}$$

mir $r_0 = n$. Wegen $r_i \in \mathbb{N}_0$ folgt so: Der Algorithmus terminiert nach spätestens n Durchläufen. Eine solche strikt monoton fallende Größe nennt man *(Schleifen-)Variante*.

- *Definierte Schritte:* Einzige „Schwierigkeit“: Division mit Rest. Hierzu existiert ein „Elementaralgorithmus“ (siehe Pascal-Standard).

- *Definitheit:* Zu zeigen: $n, m \in \mathbb{N}$ gilt stets in Schritt 1 (ansonsten ist Division mit Rest nicht definiert).
  Beweis: Richtig für Eingabe („nach 0. Durchlauf“): Gilt $n, m \in \mathbb{N}$ in Schritt 3 im i-ten Durchlauf, so gilt im $(i+1)$-ten Durchlauf:

  $$m = n \cdot q + r, \quad 0 \leq r < n, \quad r \in \mathbb{N}_0$$

  Ist $r = 0$, so terminiert der Algorithmus in Schritt 2, ansonsten ist $r \in \mathbb{N}$, $n \in \mathbb{N}$ und damit in Schritt 3 im $(i+1)$-ten Durchlauf auch $m, n \in \mathbb{N}$.

- *Effizienz:* Speicher : 4 natürliche Zahlen, Rechenzeit : „nicht schlecht“ (besser als bei Ausnutzung der Invarianten $ggt(m, n) = ggt(m\text{-}n, n)$

1.2 Fallstudien zur Softwarequalität (oder -misere?)

Fehlerhafte Spezifikation/Korrektheit/Definitheit bei:

- Untergang der „Sleipner A“ Ölplattform
- Verlust des „Mars Climate Orbiter“
- Fluggesellschaftsroutenbuchungen auf „gut Glück“
- PLZ in Wuppertal (← fehlerhafte Spezifikation)
- neues Sommerzeitende und Terminkalender
- $\{0, \ldots, 99\}$ als Jahreszahlen
- Ampelsteuerung
- Fehlalarm im Kanaltunnel
- Berliner Magnetbahn (← fehlende Plausibilitätsüberlegungen)
- Ausfall der „Telefon“-Computer
- Glücksspiel und „Zufallszahlen“
- AOL offline
- DB bucht doppelt
- falsche Telefontarife (← Image-Verlust, Klage gegen SW-Lieferanten)
- Postbank: falsche Zinsberechnung
- Flughafen Düsseldorf-Luftfrachtzentrum: $\geq$ 70.000 DM für Aushilfen
- Einschaltquote GfK
- Geschlossene Türen in Glasgow (← mechanische Notsysteme? Konsistenzüberlegungen?)
- THERAC-25
- Flugzeug-Schleudersitz
- A 320 in Warschau: Bremssystem zu intelligent?

- A 300 in China : Copilot gegen SW ($\rightarrow$ 264 Tote)
- Ungenügende Unsicherheitsinfos von errechneten Ergebnissen: AEGIS
- Rundungsfehler und die Raketenabwehr im Irak-Krieg
- Wann ist $1.407\ldots = 0.64$?
- Pentium FPU-Fehler (HW-Spezifikationsfehler oder fehlende Tests?)
- Ariane 5 Explosion

1.3 Qualitätsanforderungen an Software

1.3.1 Produktorientierte Gütekriterien

1. Funktionale Korrektheit
2. Funktionale Vollständigkeit
3. Robustheit gegenüber dem Benutzer
4. Benutzerfreundlichkeit
5. Effizienz in Laufzeit
6. Effizienz im Arbeitsspeicherbedarf
7. Effizienz im Plattenplatzbedarf
8. Integrität (gegenüber unautorisierten Änderungen)
9. Kompatibilität/Integrationsfähigkeit/Erfüllen von Standards

1.3.2 Projektorientierte Gütekriterien

1. Überprüfbarkeit
2. Verständlichkeit
3. Wartbarkeit
4. Änder- und Erweiterbarkeit
5. Portierbarkeit
6. Wiederverwendbarkeit insbesondere von Teilproblemlösungen

1.3.3 Spezifikation und Verifikation von Code

```
////////////////////////////////
// Datei:   power.cc
// Version: 1.0
// Zweck:   while-Schleife
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

#include        <iostream>
#include        <iomanip>

using namespace std;

double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
        if ((exp % 2 ) != 0) {
                erg *= x;
                exp--;
        } else {                // hier ist exp gerade
                x = x*x;
                exp = exp/2;
        }
  };
  return erg;
};

int main()
{
 cout << setprecision(10) << power2(13.5, 3) << endl;

 return 0;
}
```

Schleifeninvariante, -variante, Vor- und Nachbedingung ...

1.3.4 Explizite und implizite Spezifikation von Funktionen

Funktionen können implizit

$$
\begin{array}{ll}
1.0 & max\,(s : \mathbb{N}_1\text{-}\mathsf{set})m : \mathbb{N}_1 \\
.1 & \mathsf{pre}\ \ \mathsf{card}\ s \neq 0 \\
.2 & \mathsf{post}\ m \in s \wedge \forall\, x \in s \cdot m \geq x
\end{array}
$$

oder explizit spezifiziert werden.

$$
\begin{array}{ll}
2.0 & max : \mathbb{N}_1\text{-}\mathsf{set} \to \mathbb{N}_1 \\
.1 & max\,(s) \triangleq \\
.2 & \quad \iota\, m \in s \cdot \forall\, x \in s \cdot m \geq x \\
.3 & \mathsf{pre}\ \ \mathsf{card}\ s \neq 0
\end{array}
$$

Diskutieren sie Vor- und Nachteile.

1.4 Konstruktive Maßnahmen zur Erreichung von Softwarequalität

1. Konstruktive Voraussicht und methodische Restriktion
2. Strukturierung
3. Modularisierung
4. Lokalität
5. Integrierte Dokumentation
6. Standardisierung
7. Funktionale und informale Bindung
8. Schmale Datenkopplung
9. Vollständige Spezifikation und Verifikation / Tests
10. Lineare Kontrollstrukturen
11. Verbalisierung bei Vergabe von Namen
12. Objektorientiertes Design

1.5 Datenkapseln und formale Spezifikation - ein Beispiel

1.5.1 Die formale Spezifikation eines Moduls

module $Queue\text{-}Module$

definitions

types

3.0 X is not yet defined;

4.0 $Message = \text{OK} \mid \text{THE QUEUE IS EMPTY} \mid$
.1 THE QUEUE IS FULL

values

5.0 $QueueMax : \mathbb{N}$ is not yet defined

6.0 state $Queue\text{-}Module$ of
.1 $Queue : X^*$
.2 inv $mk\text{-}Queue\text{-}Module(Queue) \triangleq$
.3 $0 \leq \text{len } Queue \wedge \text{len } Queue \leq QueueMax$
.4 init $mk\text{-}Queue\text{-}Module(Queue) \triangleq$
.5 $Queue = [\,]$
.6 end

operations

7.0 $AddToQueue\,(i : X)$Report:Message
.1 ext wr $Queue : X^*$
.2 pre len $Queue < QueueMax$
.3 post $Queue = \overleftarrow{Queue} \curvearrowright [i] \wedge$
.4 $Report = \text{OK}$
.5 errs QUEUE-FULL : $(\text{len } Queue = QueueMax) \rightarrow$
.6 $Queue = \overleftarrow{Queue} \wedge$
.7 $Report = \text{THE QUEUE IS FULL}$

;

8.0 *DeleteFromQueue* ()Report:Message
.1 ext wr $Queue : X^*$
.2 pre $0 < \mathsf{len}\ Queue$
.3 post $Queue = \mathsf{tl}\ \overleftarrow{Queue} \wedge$
.4 $Report = \mathrm{OK}$
.5 errs QUEUE-EMPTY : $(\mathsf{len}\ Queue = 0) \rightarrow$
.6 $Queue = \overleftarrow{Queue} \wedge$
.7 $Report =$ THE QUEUE IS
.8 EMPTY

;

9.0 *Front* ()i:X, Report:Message

⋮

;

10.0 *Queue-Empty* ()i:$\mathbb{B}$, Report:Message

⋮

end *Queue-Module*

informell: FIFO = „first in first out“ (gerechtes Anstellen)

Diese Spezifikation in VDM (vienna development method, [Jon90]) ist eine in mathematischen Begriffen „beschreibende“ Spezifikation von Problemen/Algorithmen/Datenstrukturen/Datenkapseln (ADT's).

1.5.2 Struktur von Modulen

Generizität *oder* die Wiederverwendung strukturgleicher Datenkapseln (Queue von char, Queue von Number, ...)

module *Queue-Module*

parameters

types

11.0 X;

values

12.0 $QueueMax$;

imports

13.0 from *Types-Module*

.1 Message;

exports

types

14.0 $Types\text{-}Module'Message$;

.1 $Queues$;

operations

15.0 $AddToQueue : X \xrightarrow{o} Message$

.1 $DeleteFromQueue : () \xrightarrow{o} Message$

.2 $Front : () \xrightarrow{o} X \times Message$

.3 $Queue\text{-}Empty : () \xrightarrow{o} \mathbb{B} \times Message$

definitions

types

16.0 $Queues = X^*$

17.0 state *Queue-Module* of

.1 $myQueue : Queues$

.2 inv ... $\underline{\triangle}$ ...

.3 end

end *Queue-Module*

Benutzung:

module $Module\text{-}Usage$

instantiation

18.0 $NumQueue$ as $Queue\text{-}Module\,(X \to Numbers,$
.1 $QueueMax \to$
.2 $NumQueuePopulation),$

19.0 $CharQueue$ as $Queue\text{-}Module\,(X \to string,$
.1 $QueueMax \to$
.2 $CharQueuePopulation)$

definitions

20.0 state $Queue\text{-}Usage$ of
.1 $s1 : NumQueue'Queues;$
.2 $s2 : CharQueue'Queues;$
.3 inv $mk\text{-}Queue\text{-}Usage(s1, s2) \triangleq$
.4 $0 \leq \mathsf{len}\ s1 \wedge \mathsf{len}\ s1 \leq NumQueuePopulation$
.5 $0 \leq \mathsf{len}\ s2 \wedge \mathsf{len}\ s2 \leq CharQueuePopulation$
.6 init $mk\text{-}Queue\text{-}Usage(s1, s2) \triangleq$
.7 $s1 = [\,] \wedge s2 = [\,]$
.8 end

operations ...

end $Module\text{-}Usage$

mit (zuvor) definierten

definitions

types

21.0 $number = \mathbb{N};$

22.0 $string = char^{+};$
.1 ...

values

23.0 $NumQueuePopulation : \mathbb{N} = 100;$

24.0 $CharQueuePopulation : \mathbb{N} = 100;$
.1 ...

1.5.3 Sequenzen und andere Notationen in VDM

$X^*,\ X^+$
$[\,],\quad [\mathrm{i}],\quad [n^2 \mid n \in \{1 \ldots 5\}]$
$[n^2 \mid n \in \{1 \ldots 5\}](\mathrm{i})$
$\mathrm{v} \in X^* : \mathrm{v}(\mathrm{i})$
$\mathsf{len}\ , \curvearrowright$
$\mathsf{tl}\ ,\quad \mathsf{hd}$
$=,\quad \neq$
$\mathsf{inds}\ = \mathsf{dom}$
$\mathsf{elems}\ = \mathsf{rng}$

Anderes:

$\mathbb{B}$
$\neg\, E_1$
$E_1 \wedge E_2,\quad E_1 \vee E_2$
$E_1\ \Rightarrow\ E_2$
$E_1\ \Leftrightarrow\ E_2$
$\forall x \in S \cdot E,\quad \forall x : T \cdot E$
$\exists\, x \in S \cdot E\quad \exists\, x : T \cdot E,\quad \exists!\, x \in S \cdot E$

1.5.4 Compilationseinheiten (Programme und Module) in JAVA

(Datenkapsel = Klasse)

```
class ModuleName{
     public static void main(String[] args){
            System.out.println("Dies ... ");
            // ...
     }
}
```

Hier noch ein weiteres Beispiel für ein Stück JAVA-Code:

```
class ModuleName{
      private static int ModuleId;
      private int x;
      public int x(){
            return x;
      }
      public void setX(int newX){
            x=newX;
      }
}

...

ModuleName myX = new ModuleName();
myX.setX(1);
System.out.println( "x=" + myX.x() );
```

Die einzelnen Schlüsselwörter haben folgende Bedeutungen:

- `private` nur diese Klasse
- `protected` nur Unterklassen
- `public` alle Klassen
- `package` nur Klassen dieses Packages

1.5.5 ADT's / Datenkapseln

Datentypen können i.a. durch

- eine *Menge von Werten*, die irgendwie durch Symbole benannt werden, und
- eine *Menge von Operationen* auf den Werten, die entweder explizit durch „Wertetabellen" der Operationen oder durch *eine ausreichende Menge von Gesetzen*, die durch die Operationen erfüllt werden sollen,

beschrieben werden. Betrachten wir ein Beispiel:

1.5.6.1 Axiomatische Spezifikation

Der abstrakte Datentyp (ADT) Boolean soll zwei Werte besitzen, die mit den Symbolen „wahr" und „falsch" bezeichnet werden sollen:

W(Boolean) := {true, false} (Wertebereich, entspricht o(Boolean))

Auf B:= W(Boolean) sollen die Operationen

O(Boolean):= {*true*: $B^0 \to B$ (Konstante aus B);
not: $B \to B$;
and, *or*, *equiv*, *imp* : $B \to B$;
ifthenelse: $B \times (B \times B) \to B$ }

mit folgenden (definierenden) Gesetzmäßigkeiten zur Verfügung stehen:

Axiomatische Def. (Boolean):	v. S.
`true() = true`	`true`
Idempotenz: `not(not(x)) = x` $\forall x \in B$	
Nichttrivialität: `not(true)` $\neq$ `true`	`not`
Assoziativität: `and(and(x,y),z) = and(x,and(y,x))` $\forall x, y, z \in B$	
Kommutativität: `and(x,y) = and(y,x)` $\forall x, y \in B$	

Ausschluß: `and(x,not(x)) = not(true)` $\forall x \in B$
`and(true,true) = true`
`and(not(true),not(true)) = not(true)`
	and

abgeleitete Operationen:	
$\forall x, y \in B$: *`or(x,y)`* `= not(and(not(x),not(y)))`	or
`equiv(x,y)` `= or(and(x,y),` `and(not(x),not(y)))`	equiv
`impl(x,y)` `= or(not(x),y)`	impl
$\forall x, y, z \in B$ *`ifthenelse`*`(x,y,z) =` $\begin{cases} \texttt{y}, \texttt{falls} \\ \quad \texttt{x} = \texttt{true} \\ \texttt{z}, \texttt{falls} \\ \quad \texttt{x} = \texttt{not}(\texttt{true}) \end{cases}$	ifthenelse

Bemerkung:
Die Abkürzung *v.S.* in der Tabellenüberschrift steht für *„vollständige Spezifikation“*.

1.5.6.2 Implementierungen

Realisiert werden kann dieser Datentyp durch sehr verschiedene Modelle:

1. Modell „Aufzählungstyp mit Wertetabellen“:

```
Modulimplementierung Boolean:

  type
        Boolean = {true,false};

  function not(w: Boolean): Boolean;
  begin

          if (w = true) then
               return false
          else return true;

  end;

  function and(l,r: Boolean): Boolean;
  begin

          if (l = true) then
               if (r = true) then return true;
          return false;

  end;

  function or(l,r: Boolean): Boolean;
  begin

          return not(and(not(l),not(r)));

  end;

end;
```

Klasse eines Aufzählungstyps:

```
////////////////////////////////////////////
// Datei:   enum_Day5.cc
// Version: 5.0
// Zweck:   enum-Klasse, >> mit Zurückschreiben, cerr-Warnung
// Autor:   Hans-Juergen Buhl
// Datum:   23. Nov. 99
////////////////////////////////////////////

#include        <iostream>
#include        <sstream>
#include        <string>

using namespace std;

class Day{
private:
        enum DayType {_Montag, _Dienstag, _Mittwoch, _Donnerstag,
                      _Freitag, _Samstag, _Sonntag};

// nicht als member zugelassen; nur Konstruktoren dürfen als
//    Initialisierer von Membern in Klassen dienen!!!
//
// static const string DayTable[] = {"Montag", "Dienstag", "Mittwoch",
//                      "Donnerstag", "Freitag", "Samstag", "Sonntag"};

        static const string DayTable[7];

        DayType t;

        Day(const DayType& dt): t(dt) {};

public:

        static const Day Montag;
        static const Day Dienstag;
        static const Day Mittwoch;
        static const Day Donnerstag;
        static const Day Freitag;
        static const Day Samstag;
        static const Day Sonntag;

        Day(const Day& d = Montag): t(d.t) {};
```

```
        Day& operator++();
        const Day operator++(int);

        friend istream& operator>>(istream&, Day&);
        friend ostream& operator<<(ostream&, const Day&);

};

const Day Day::Montag(_Montag);
const Day Day::Dienstag(_Dienstag);
const Day Day::Mittwoch(_Mittwoch);
const Day Day::Donnerstag(_Donnerstag);
const Day Day::Freitag(_Freitag);
const Day Day::Samstag(_Samstag);
const Day Day::Sonntag(_Sonntag);

const string Day::DayTable[] = {"Montag", "Dienstag", "Mittwoch", "Donnerstag",
                                "Freitag", "Samstag", "Sonntag"};

Day& Day::operator++()
{
        (*this).t = (_Sonntag == t) ? _Montag : DayType((*this).t + 1);
        return *this;
}

const Day Day::operator++(int)
{
        Day old_value(*this);
        ++(*this);
        return old_value;
}

istream& operator>>(istream& is, Day& d)
{
        string s;
        is >> s;
        for (int i = 0; i < 7; i++)
                if (s == Day::DayTable[i]) {
                        d.t =  Day::DayType(i);
                        return is;
                }
```

```
        ////////////////////////////////////////////////////////////
        // falscher Eingabestring:
        // evtl. zeichenweise mittels is.putback(ch) s zurück!

        is.putback(' ');
        for (int i = s.length()-1; i >= 0; i--)
                is.putback(s[i]);

        // is.putback(' ');
        // for (string::const_iterator i = s.end()-1; i >= s.begin(); i--)
        //      is.putback(*i);

        // is.putback(' ');
        //for (string::const_reverse_iterator i = ((const string)s).rbegin();
        //     i < ((const string)s).rend(); i++)
        //              is.putback(*i);

        ////////////////////////////////////////////////////////////

        is.clear(ios_base::badbit);
        return is;
}

ostream& operator<<(ostream& os, const Day& d)
{
        os << Day::DayTable[int(d.t)];
        return os;
}


int main()
{

// ...

}
```

2. Modell „Rückführung auf Integer“:

```
Modulimplementierung Boolean;

  type
        Boolean = 0..1;

  const

        true  = 1;
        false = 0;

  function not(w: Boolean): Boolean;
  begin

     return (1-w);

  end;

  function and(l,r: Boolean): Boolean;
  begin

     return (l*r);

  end;

  function or(l,r: Boolean): Boolean;
  begin

     return not(and(not(l),not(r)));

  end;                ↖ oder effektiver  return (r+l-r*l)

end;
```

Eine Subrange-Klasse:

```
template<int low, int high>
class Subrange{

  int value;

public:

  Subrange(const int i = 0){
    if ((i < low) || (i > high)) throw "Subrange-Ausnahmebedingung in Konstruktor";
    value = i;
  }

  // operator int() { return value; }
  int get_value()const { return value; }

  Subrange & operator = (const int i){
    if ((i < low) || (i > high)) throw "Subrange-Ausnahmebedingung in Zuweisung";
    value = i;
    return *this;
  }

};
```

Warum kann hier im Konstruktor nicht die Initialisierung mittels
`Subrange(const int i = 0): value(i) {}`
benutzt werden?

Welchen Nachteil im Sinne „abstrakter Datenkapseln“ erkaufen Sie sich, wenn Sie den Operator `int()` programmieren? Welche Vorteile bringt dieser Operator verglichen mit der Benutzung von `get_value()`?

Schreiben Sie einen Operator `Subrange operator + (const Subrange r)`.

3. Modell „Mengen“:

```
Modulimplementierung Boolean;

  type Boolean = set of 1..1;

  const

     true  = Boolean{1};
     false = Boolean{};

  function not(w: Boolean): Boolean;
  begin

     if (1 in w) then
          return false
     else return true;

  end;

  function and(l,r: Boolean): Boolean;
  begin

     return l*r;              (* l ∩ r *)

  end;

  function or(l,r: Boolean): Boolean;
  begin

    return l+r;               (* l ∪ r *)

  end;

end;
```

Hinweise:

`C: bitweise int-Operationen`

`C++/STL: bitset<32>`

`C++/STL: set<int>, multiset, ...`

Für den Benutzer und den Datentyp „Boolean“ aufrufende Programmteile ist einzig und allein

W(Boolean), d.h. die Konstanten *true*, *false*
O(Boolean) und *AxiomatischeDef (Boolean)*

von Bedeutung. Er sollte also unabhängig von der gewählten Implementierung arbeiten können.
Auch bei nachträglicher Änderung des Datentypmodells sollte bestehende Software **nicht** geändert und **nicht** neu übersetzt werden müssen. (Je nach Laufzeitsystem ist evtl. ein (explizites) neues Binden notwendig.)

- Abstrakte Datentypen werden durch Modelle realisiert, die die in der entsprechenden Sprachumgebung vorhandenen

  einfachen Datentypen

  sowie

  Typkonstruktoren

  benutzen.

- Definieren Sie bei der Problemlösung zu spezifizierende „neue“ Datentypen **nie** durch eine spezielle Implementierung, sondern durch Angabe „des ADT's“:

  W(Typ); O(Typ) mit AxiomatischerDef(Typ)
  (Konstanten, Operationen mit Eigenschaften).

1.5.6 Axiomatische Spezifikation in OBJ

Einige einfache mathematische Spezifikationen zum Kennenlernen der Sytax von OBJ:
Semigroup:

```
SPEC Semigroup
SORTS s
OPS   *  :  s s  → s
FORALL  m1, m2, m3 : s
AXIOMS
        (1) (m1 * m2) * m3 = m1 * (m2 * m3)
ENDSPEC
```

Monoid:

```
SPEC Monoid
USING Semigroup
OPS   e :   → s
FORALL m : s
AXIOMS
       (1) e * m = m
       (2) m * e = m
ENDSPEC
```

Group:

```
SPEC Group
USING Monoid
OPS    ()^-1:  s → s
FORALL m : s
AXIOMS
       (1) m * m^-1 = m
       (2) m^-1 * m = m
ENDSPEC
```

Ring:

```
SPEC Ring
USING Semigroup
OPS    z :      → s
       + : s s → s
       - :   s → s
FORALL m1, m2, m3 : s
AXIOMS
       (1) (m1 + m2) + m3 = m1 + (m2 + m3)
       (2) m1 + m2 = m2 + m1
       (3) m1 + z = m1
       (4) m1 + (-m1) = z
       (5) (m1 + m2) * m3 = (m1 * m3) + (m2 * m3)
       (6) m1 * (m2 + m3) = (m1 * m2) + (m1 * m3)
ENDSPEC
```

```
SPEC  Queue

USING Natural + Boolean

SORT  queue

OPS

      new         : → queue
      add         : queue nat → queue
      remove      : queue → queue
      front       : queue → nat
      is-empty?   : queue → bool
      queue-error : → queue
      nat-error   : → nat


FORALL

       q : queue
       n : nat

AXIOMS

      (1) front(new) = nat-error

      (2) remove(new)= queue-error

      (3) is-empty?(new) = true

      (4) is-empty?(add(q,n)) = false

      (5) front(add(q,n))  = IF is-empty?(q) THEN n
                             ELSE front(q) ENDIF

      (6) remove(add(q,n)) = IF is-empty?(q) then new
                             ELSE add(remove(q),n) ENDIF

ENDSPEC
```

Vgl. dazu [Tur94], Kap. 8 ff.

Bintree:

```
SPEC Bintree(data)
USING data + nat + bool
SORTS Bintree
OPS
        Leaf  : data         → Bintree
        Left  : Bintree data → Bintree
        Right : data Bintree → Bintree
        Both  : data Bintree data → Bintree

        Breadth, Edge, Node : Bintree → nat
        Deg                 : Bintree → bool
FORALL
        a : data
        b, b1, b2 : Bintree
AXIOMS
        (1) Breadth(Leaf(a))    = succ(0)
        (2) Breadth(Left(b,a))  = Breadth(b)
        (3) Breadth(Right(a,b)) = Breadth(b)
        (4) Breadth(Both(b1,a,b2) = ADD(Breadth(b1),Breadth(b1))

        (5) Edge(Leaf(a))    = 0
        (6) Edge(Left(b,a))  = succ(Edge(b))
        (7) Edge(Right(a,b)) = succ(Edge(b))
        (8) Edge(Both(b1,a,b2)) = SUCC(SUCC(ADD(Edge(b1),Edge(b2))))

        (9) Node(b) = SUCC(Edge(b))

        (10) Deg(Leaf(a))    = true
        (11) Deg(Left(b,a))  = Deg(b)
        (12) Deg(Right(a,b)) = Deg(b)
        (13) Deg(Both(b1,a,b1)) = false
ENDSPEC
```

1.5.7 Problem der Implementierungsabhängigkeit

Ausschnitt aus einem Programm zur Berechnung konvexer Hüllen endlich vieler Punkte des $\mathbb{R}^2$:

```
i := kh.f[kh.b];

kh.d[i].a := 1;

kh.d[i].e := 1;
kh.vor[i] := 1;

     ⋮

vorl := s[kh.d[kh.vor[i]].a];
```

Bemerkung:

- schlecht lesbar (Verbalisierung?)
- Implementierungsänderung von Feld zu Zeiger ohne „Aufwand“ unmöglich

1.6 Umgangssprachliche Spezifikation?

1.6.1 Das Parkplatzproblem

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

„Lösung“: Sei

$$\begin{array}{rcl} P & := & \text{Anzahl der PKW's} \\ M & := & \text{Anzahl der Motorräder} \end{array}$$

$$\left\{ \begin{array}{rcl} P + M & = & n \\ 4P + 2M & = & m \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} M & = & n \text{ - } P \\ P & = & \frac{m \text{ - } 2n}{2} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} M & = & \frac{4n \text{ - } m}{2} \\ P & = & \frac{m \text{ - } 2n}{2} \end{array} \right\}$$

„Algorithmus“:

```
        ⋮

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

        ⋮
```

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

Besser ist also:

Eingabe : $m, n \in \{0, 1, \ldots, \text{INT\_MAX}\}$

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

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

Nachbedingung : Ein $(P, M) \in \{0, 1, \ldots, \text{INT\_MAX}\}$ mit

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

1.7 Wahrheitstabellen und Entscheidungstabellen

1.7.1 Wahrheitstabellen

`not` : $\mathbb{B} \to \mathbb{B}$

x	not x
true	false
false	true

`or` : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$

x/y	true	false
true	true	true
false	true	false

`and` : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$

x/y	true	false
true	true	false
false	false	false

`ifthenelse` : $\mathbb{B} \times (\mathbb{B} \times \mathbb{B}) \to \mathbb{B}$
$(x, y, z) \mapsto$ `ifthenelse(x,y,z)`

`x=true`

y/z	true	false
true	true	true
false	false	false

`x=false`

y/z	true	false
true	true	false
false	true	false

Definition 1.7.1
Eine Menge heißt *aufzählbar*, wenn man ihre Elemente durch einen Algorithmus nacheinander erzeugen kann oder wenn sie leer ist;
formaler: $M \subseteq G$ heißt *aufzählbar* $:\Leftrightarrow M = \{\}$ oder $\exists f \cdot : \mathbb{N} \to G$ mit $f(\mathbb{N}) = M$

Die Spezifikation findet hier also durch Aufzählung aller (Urbild,Bild)-Paare statt.

1.7.2 Entscheidungstabellen

Entscheidungstabellen sind in der kommerziellen DV weit verbreitet. Es handelt sich um eine wartungsfreundliche, d.h. lesbare graphische Darstellung zur Überprüfung aller Kombinationsmöglichkeiten in verschachtelten `if-then-else`-Anweisungen:

`Bed1`	w_{11}	w_{12}	$\ldots$	w_{1k}
`Bed2`	w_{21}	w_{22}	$\ldots$	w_{2k}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
`BedN`	w_{N1}	w_{N2}	$\ldots$	w_{Nk}
`Anwg1`	f_{11}	f_{12}	$\ldots$	f_{1k}
`Anwg2`	f_{21}	f_{22}	$\ldots$	f_{2k}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
`AnwgP`	f_{P1}	f_{P2}	$\ldots$	f_{Pk}

$w_{ij} \in \{t, f, \text{-}\}$

$f_{ij} \in \{\mathtt{x}, \cdot\}$

Erklärung der auftretenden Symbole:

t	$\hat{=}$	`true`
f	$\hat{=}$	`false`
-	$\hat{=}$	beliebig, d.h. `true` oder `false`
x	$\hat{=}$	Ausführung
$\cdot$	$\hat{=}$	Nicht-Ausführung

Anwendung der Entscheidungstabelle:

Berechne den Wert von

$$\begin{pmatrix} Bed1 \\ \vdots \\ BedN \end{pmatrix} =: b$$

im aktuellen Status. Suche Spalte

$$\begin{pmatrix} w_{1j} \\ \vdots \\ w_{Nj} \end{pmatrix},$$

die zu b paßt. Führe dann alle Anweisungen nacheinander aus, für die $f_{1j} = \mathtt{x}$.

a) Es darf in jedem Status nur genau eine Spalte geben, für die b paßt.

b) Zu jedem möglichen Wert von b sollte eine passende Spalte f existieren.

Entscheidungstabellen sind direkt in `Algol 68` vorhanden oder sind durch Precompiler in beliebige Sprachen übersetzbar!

Beispiel:

B1:	Konto überzogen?	t	t	f	f
B2:	Bisheriges Zahlungsverhalten einwandfrei?	t	f	t	f
A1:	Scheck einlösen	x	·	x	x
A2:	Scheck nicht einlösen	·	x	·	·
A3:	Kredit einräumen	x	·	·	·

oder kürzer

B1:	Konto überzogen?	t	t	f
B2:	Bisheriges Zahlungsverhalten einwandfrei?	t	f	-
A1:	Scheck einlösen	x	·	x
A2:	Scheck nicht einlösen	·	x	·
A3:	Kredit einräumen	x	·	·

Vgl. dazu [Str77].

1.8 Datenstrukturen und „unendlich“ große „Datensammlungen“

1.8.1 Datenstruktur

Der Begriff *Datenstruktur* wird häufig als Synonym für *Datentyp* benutzt. Eigentlich (und in dieser Vorlesung) bezeichnet er sogenannte

Container-Datentypen

Das sind Datentypen, deren *Objekte* (Exemplare) eine Ansammlung von Daten - meist gleichen Typs - mit gewissen Beziehungen untereinander enthalten.

Beispiele:
Warteschlange, Feld, Verbund, Datei, ...
mit linearer Beziehung der Elemente zueinander.

Eine Datenstruktur wird mit Hilfe von Typkonstruktoren aus elementaren Datentypen (integer $\neq \mathbb{Z}$, double $\neq \mathbb{R}$, boolean $= \mathbb{B}$, character, Aufzählungstypen, Unterbereichstypen) oder bereits zuvor definierten Datenstrukturen zusammengesetzt.

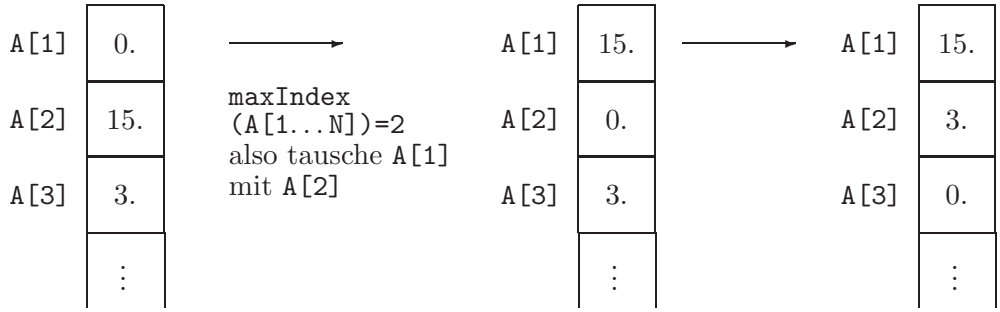
Typkonstruktoren:

Aggregation	(kartesisches Produkt)	`Feld, Klasse`
Generalisation	(disjunkte Vereinigung)	`union`
allg. Sequenzbildung	(endl., aber beliebig lange Sequenzen)	`file` ... Verkettung mit Zeigern
Potenzmengenbildung	$\mathbb{P}$	`map` ...
Funktionsbildung	$(\rightarrow)$	`function`
Rekursion		Verkettung mit Zeigern

1.8.2 Umsortierung, Indizes, Kettung mit Cursor

1.8.2.1 Laufzeitungünstiges Umsortieren von Feldern

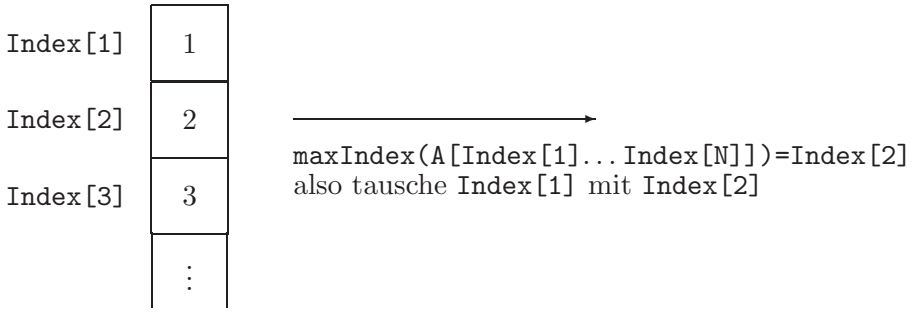
Statt

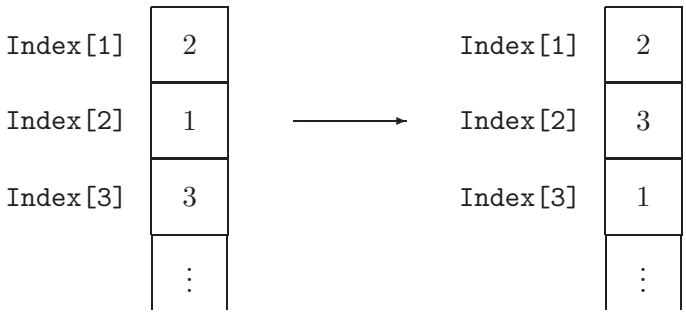


wird `A` *unverändert erhalten, wenn man eine Indizierung vornimmt.*

1.8.2.2 Indizierte Felder

Man geht folgendermaßen vor :





Schließlich:

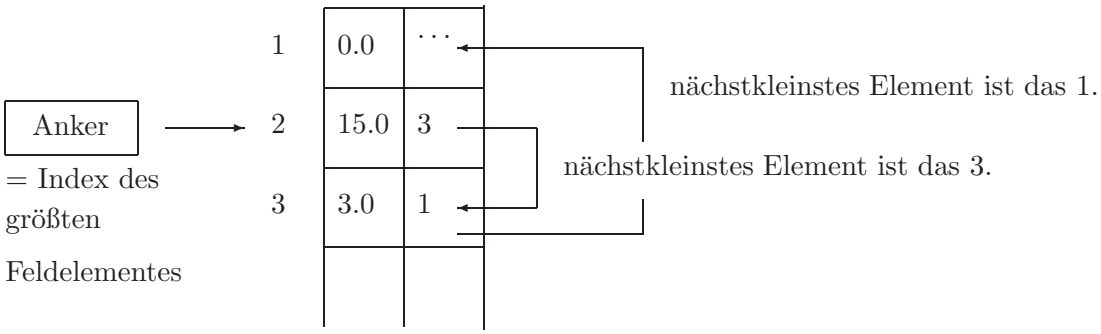
$$\texttt{A[Index[1]]} \geq \texttt{A[Index[2]]} \geq \texttt{A[Index[3]]} \geq \ldots$$

(Ganzzahlige Variablen, die als Index in ein Feld fungieren heißen *Cursor*.)

+ laufzeitgünstiges Sortieren ⇒ Index-Feld
+ mehrere Indizes möglich
- Problem : Änderung des Feldes zur Laufzeit, z.B. Löschen eines Elementes!

1.8.2.3 Kettung

Ausweg: Jeder Feldeintrag enthält neben seinem Wert den Index dex „nächstfolgenden“ Feldeintrages.



+ laufzeitgünstiges Sortieren, da nur Index-Werte getauscht werden
+ laufzeitgünstiges Löeschen, ...
- $\mathcal{O}(n)$-Aufwand, um auf das n-te Element zuzugreifen

1.8.3 Potentiell unendlich große Datenstrukturen

Mittels

```
template<class T>
class Queue{

  T Inhalt[QueueMax];

  Subrange<-1,QueueMax-1> Start;
  Subrange<0,QueueMax-1> Ende;

  // Start == -1   <=>  leer; sonst:
  // Start <= Ende:    Queue == Inhalt[Start ... Ende]
  // Start >  Ende:    Queue == Inhalt[Start ... QueueMax-1, 0 ... Ende]
  //
  // Ringpuffer, um unnötiges Vershieben zu vermeiden

  ...
}
```

werden statische Datenobjekte explizit kontrollierbarer Lebenszeit erzeugt (*creator/constructor*

```
Queue<int> q_int;
```

)

Da im „statischen Text“ eines Algorithmus nur endlich viele Konstruktoren stehen können, ist die Erzeugung von *potentiell unendlich vielen* (d.h. endlich viele, aber zur Laufzeit bestimmt viele) Daten (als Objekte/Komponenten eines Containertyps) nur auf zwei Arten möglich:

a) Rekursiver Selbstaufruf einer

```
Queue<int> q_int;
```

enthaltenden Routine.

b) Benutzung von dynamischen Konstruktoren, die intern `new` benutzen.

Da der rekursive Selbstaufruf i.a. unnötigen Overhead verursacht und kein Zugriff auf die lokalen Variablen aufrufender Prozeduren erfolgen kann, zieht man in der Praxis die Methode b) vor:

Beispiel:
Will man einen `ADT` *Queue* spezifizieren, deren Objekte (die Warteschlangen) als Komponenten von potentiell undendlich langen Datenstrukturen auftreten können sollen, so benötigt man etwa ein Modul `ManageQueues`. Es stellt beliebig „viele“ Queues bereit, durch die dynamische Erzeugung von Queues etwa in einer Schleife mittels „`x:=makeQueue();`“

```
        module ManageQueues
          operations
25.0          makeQueue: () --o--> Ids × Message;
26.0          Front: Ids → X × Message;
27.0          QueueEmpty: Ids → 𝔹 × Message;
28.0          AddToQueue: X × Ids → Message;
29.0          DeleteFromQueue: Ids → Message;
30.0          RemoveQueue: Ids → Message
        end ManageQueues
```

1.8.4 Spezifikation und Implementierung von Datentyp-Managern

1.8.4.1 Der Status von `ManageQueues`

module $ManageQueues$

definitions

types

31.0 X is not yet defined;

32.0 $XQueues = X^*$

values

33.0 $QueuesMax : \mathbb{N}$ is not yet defined

34.0 state $ManageQueues$ of
.1 $Queues : \mathsf{token} \xrightarrow{m} XQueues$
.2 inv $mk\text{-}ManageQueues(Queues) \triangleq$
.3 $t \mapsto q \in Queues \Rightarrow$
.4 $0 \leq \mathsf{len}\ q \wedge \mathsf{len}\ q \leq QueuesMax$
.5 init $mk\text{-}ManageQueues(Queues) \triangleq$
.6 $Queues = \{\mapsto\}$
.7 end

1.8.4.2 Operationen von `ManageQueues`

operations

35.0 $makeQueue\,()$t:[token],Report:Message
.1 ext wr $Queues : \mathsf{token} \xrightarrow{m} XQueues$
.2 post $t \in \mathsf{token} - \mathsf{dom}\ \overleftarrow{Queues} \wedge Queues =$
.3 $\overleftarrow{Queues} \dagger \{t \mapsto [\,]\} \wedge Report = \mathrm{OK}$
.4 $\vee\ t = nil \wedge Queues = \overleftarrow{Queues}$
.5 $\wedge\ Report = \mathrm{NO\ MORE\ MEMORY}$
;

36.0 $AddToQueue\,(x : X, t : \mathsf{token})$Report:Message
.1 ext wr $Queues : \mathsf{token} \xrightarrow{m} XQueues$

$$
\begin{array}{ll}
.2 & \textsf{pre}\ \ t \in \textsf{dom}\ \mathit{Queues} \wedge \\
.3 & \quad \textsf{len}\ \mathit{Queues}(t) < \mathit{QueuesMax} \\
.4 & \textsf{post}\ \mathit{Queues}(t) = \overleftarrow{\mathit{Queues}(t)} \curvearrowright [x] \wedge \\
.5 & \quad \mathit{Report} = \mathrm{OK} \\
.6 & \textsf{errs}\ \mathrm{NO\text{-}QUEUE} : (t \notin \textsf{dom}\ \mathit{Queues}) \rightarrow \\
.7 & \qquad\qquad \mathit{Queues} = \overleftarrow{\mathit{Queues}} \wedge \\
.8 & \qquad\qquad \mathit{Report} = \mathrm{QUEUE\ DOES\ NOT\ EXIST} \\
.9 & \quad \mathrm{QUEUE\text{-}FULL} : (t \in \textsf{dom}\ \mathit{Queues} \wedge \\
.10 & \qquad\qquad \textsf{len}\ \mathit{Queues}(t) = \mathit{QueuesMax}) \rightarrow \\
.11 & \qquad\qquad \mathit{Queues}(t) = \overleftarrow{\mathit{Queues}(t)} \wedge \\
.12 & \qquad\qquad \mathit{Report} = \mathrm{THE\ QUEUE\ IS\ FULL}
\end{array}
$$

;

$$
\begin{array}{ll}
37.0 & \mathit{Front}\,(t : \textsf{token})\mathrm{x{:}X,\ Report{:}Message} \\
.1 & \textsf{ext rd}\ \ \mathit{Queues} : \textsf{token} \xrightarrow{m} \mathit{XQueues} \\
.2 & \textsf{pre}\ \ t \in \textsf{dom}\ \mathit{Queues} \wedge \\
.3 & \quad 0 < \textsf{len}\ \mathit{Queues}(t) \\
.4 & \textsf{post}\ x = \textsf{hd}\ \mathit{Queues}(t) \wedge \mathit{Report} = \mathrm{OK} \\
.5 & \textsf{errs}\ \mathrm{NO\text{-}QUEUE} : (t \neq \textsf{dom}\ \mathit{Queues}) \rightarrow \\
.6 & \qquad\qquad \mathit{Queues} = \overleftarrow{\mathit{Queues}} \wedge \\
.7 & \qquad\qquad \mathit{Report} = \ldots \\
.8 & \quad \mathrm{QUEUE\text{-}EMPTY} : (t \in \textsf{dom}\ \mathit{Queues} \wedge \textsf{len}\ \mathit{Queues}(t) = 0) \\
\rightarrow & \\
.9 & \qquad\qquad\qquad \mathit{Queues} = \overleftarrow{\mathit{Queues}} \wedge \\
.10 & \qquad\qquad\qquad \mathit{Report} = \mathrm{THE\ QUEUE\ IS\ EMPTY}
\end{array}
$$

;

$$
\begin{array}{ll}
38.0 & \mathit{deleteQueue}\,(t : \textsf{token})\mathrm{Report{:}Message} \\
.1 & \textsf{ext wr}\ \ \mathit{Queues} : \textsf{token} \xrightarrow{m} \mathit{XQueues} \\
.2 & \textsf{pre}\ \ t \in \textsf{dom}\ \mathit{Queues} \\
.3 & \textsf{post}\ \mathit{Queues} = \{t\} \lhd \overleftarrow{\mathit{Queues}} \\
.4 & \textsf{errs}\ \mathrm{NO\text{-}QUEUE} : (t \neq \textsf{dom}\ \mathit{Queues}) \rightarrow \\
.5 & \qquad\qquad \mathit{Queues} = \overleftarrow{\mathit{Queues}} \wedge \\
.6 & \qquad\qquad \mathit{Report} = \ldots
\end{array}
$$

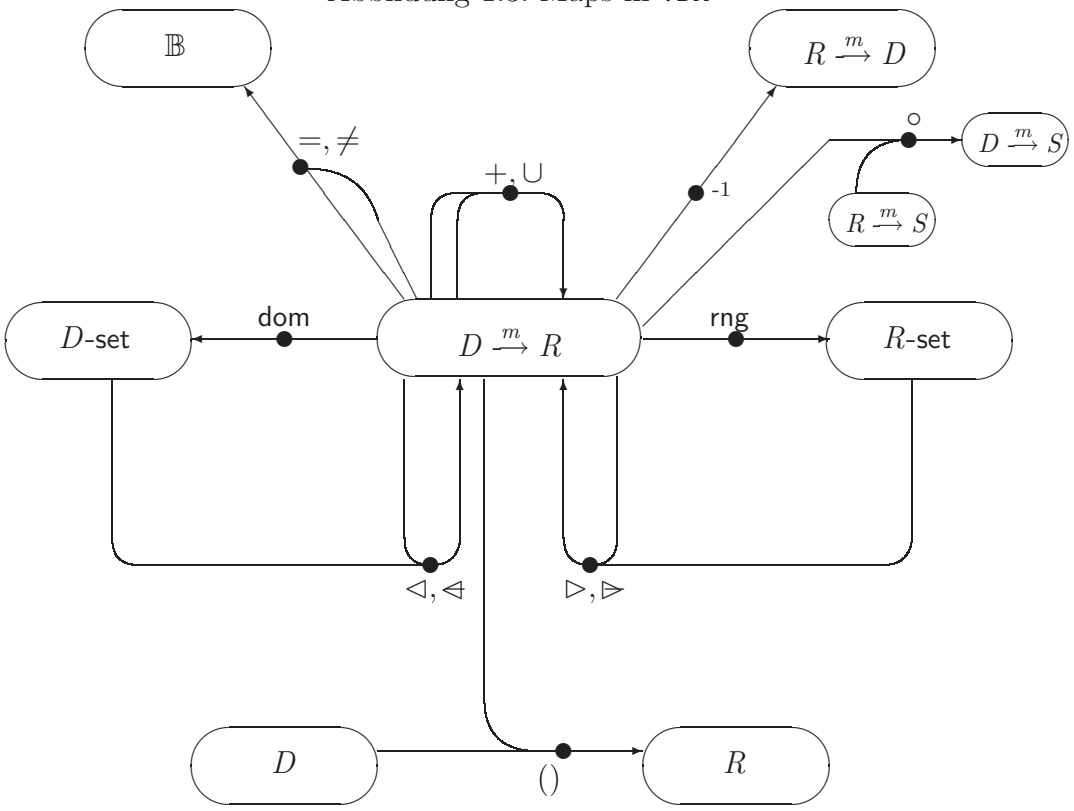
end *ManageQueues*

1.8.4.3 Implementierungstips

In C++ vorzugsweise mit Zeigern zu implementieren, vgl. Seite **??**ff.

1.8.5 Maps in VDM

Abbildung 1.3: Maps in VDM



$s \lhd m$	*domain restricted to*	$m \rhd s$	*range restricted to*
$s \lhd\!\!\!- m$	*domain restricted by excluding*	$m \rhd\!\!\!- s$	*range restricted by excluding*

Mit $M : T_1 \xrightarrow{m} T_2$ gilt:

$$M = \{t_1 \mapsto t_2 \mid t_1 \in \mathsf{dom}\ M, t_2 \in \mathsf{rng}\ M \cdot t_2 = M(t_1)\}$$
$$= \{mk\text{-}(t_1, t_2) \mid t_1 \in \mathsf{dom}\ M, t_2 \in \mathsf{rng}\ M \cdot t_2 = M(t_1)\};$$

$\{\mapsto\}$ ist die leere Map;

$M : T_1 \xleftrightarrow{m} T_2$ (bijektive Map) ist äquivalent zu
$M : T_1 \xrightarrow{m} T_2$
$inv\ M \triangleq \forall m_1, m_2 \in \mathsf{dom}\ M \cdot (M(m_1) = M(m_2)) \Rightarrow (m_1 = m_2)$;

Für $M : T_1 \xleftrightarrow{m} T_2 \quad \exists\, M^{-1} : T_2 \xrightarrow{m} T_1$
falls $\mathsf{rng}\ M$ keine Funktionen enthält.

(T_1-set bzw. $T_1 \xrightarrow{m} T_2$ sind nicht möglich für T_1, die Funktionstyp sind oder als zusammengesetzter Typ einen Funktionstyp als Komponente enthalten.)

$\{1 \mapsto 1, 2 \mapsto 6, 3 \mapsto 2\}$

$\{x + y \mapsto x \cdot y \mid x \in \{2, 3\}, y \in \{1, 5\}\}$

$M_1 \cup M_2$ für *kompatible* Maps M_1 und M_2.

(M_1, M_2 sind *kompatibel*, wenn $\forall m \in \mathsf{dom}\ M_1 \cap \mathsf{dom}\ M_2 \cdot M_1(m) = M_2(m)$.)

$M_1 \dagger M_2$

1.8.6 Asymptotische Kennzahlen für Speicherplatzbedarf und Rechenzeit

- $O(g(n))$:

Definition 1 *Seien* $f, g : \mathbb{N} \to \mathbb{R}_0^+$. *Dann nennt man* f durch g nach oben asymptotisch beim Parameterwachstum nach Unendlich beschränkt*, wenn*

$$f(n) = O(g(n)) \underset{def}{\Leftrightarrow} \exists\, c > 0, n_0 \in \mathbb{N} \cdot \forall\, n \geq n_0 \cdot f(n) \leq c\ g(n)$$

Beispiele: Die lineare Suche in einem unsortierten Datenbestand der Länge n ist von der Rechenzeit-Komplexität $O(n)$, die binäre Suche in einem sortierten Datenbestand von der Komplexität $O(\log_2 n)$. Der Speicherplatzbedarf für einen Vektor mit n Komponenten ist von der Komplexität $O(n)$, der für eine $n \times n$-Matrix von der Komplexität $O(n^2)$.
Jede Funktion der Komplexität $O(n^2)$ ist zugleich auch von der Komplexität $O(n^3)$. Deshalb ist die Schreibweise $f(n) = O(n^2)$ bzw. $f(n) = O(n^3)$ wegen der fehlenden Transitivität nicht sehr glücklich, aber leider üblich. Besser wäre die Schreibweise $f \in O(g(n))$ mit der Definition

$$O(g(n)) \underset{def}{=} \{f \mid \exists\, c > 0, n_0 \in \mathbb{N} \cdot \forall\, n \geq n_0 \cdot f(n) \leq c\ g(n)\},$$

wobei das n in $f \in O(g(n))$ bzw. in $f(n) = O(g(n))$ andeutet, daß es um die Asymptotik für $n \in \mathbb{N}, n \to \infty$ geht.

Ist $f \in O(g(n))$, so gilt auch

$$1034\ f \in O(g(n))$$

und

$$10^{-200}\ f \in O(g(n)).$$

Welche anderen Rechenregeln gelten?

Die Zugriffszeit auf jede beliebig Komponente eines Vektors ist von der Vektorlänge unabhängig; man sagt, der Zugriff braucht konstante Zeit $O(1)$. (Eigentlich müßte es nach der üblichen Konvention $O(1(n))$ mit der Funktion $1 : \mathbb{N} \to \mathbb{R}_0^+, n \mapsto 1$ heißen.)

Vorsicht vor Fehlschlüssen, die durch die $=$ statt der $\in$-Schreibweise nahegelegt werden: Besteht ein Algorithmus aus zwei sequentiell nacheinander stattfindenden Teilen jeweils der Komplexität $O(n^3)$, und kann man den zweiten Algorithmusteil ersatzlos einsparen, so spart man nicht unbedingt 50% Aufwand ...

• $\Omega(n)$:

Definition 2 *Seien* $f, g : \mathbb{N} \to \mathbb{R}_0^+$. *Dann nennt man* f durch g nach unten asymptotisch beim Parameterwachstum nach Unendlich beschränkt*, wenn*

$$f(n) = \Omega(g(n)) \underset{def}{\Leftrightarrow} \exists\, c > 0, n_0 \in \mathbb{N} \cdot \forall\, n \geq n_0 \cdot f(n) \geq c\ g(n)$$

• $\Theta(n)$:

Definition 3 *Seien* $f, g : \mathbb{N} \to \mathbb{R}_0^+$. *Dann nennt man* f und g von asymptotisch gleichem Wachstum beim Parameterwachstum nach Unendlich*, wenn*

$$f(n) = \Theta(g(n)) \underset{def}{\Leftrightarrow} f(n) = O(g(n)) \ \wedge\ f(n) = \ \Omega(g(n))$$

$$(\ \Leftrightarrow\ f(n) = O(g(n)) \ \wedge\ g(n) = \ O(f(n))\ )$$

(Hier liegt eine *scharfe* Aussage vor. Es handelt sich um eine Äqivalenzrelation.)

• $o(n)$:

Definition 4 *Seien* $f, g : \mathbb{N} \to \mathbb{R}_0^+$. *Dann nennt man* f von asymptotisch langsamerem Wachstum als g beim Parameterwachstum nach Unendlich*, wenn*

$$f(n) = o(g(n)) \underset{def}{\Leftrightarrow} \forall\, c > 0 \cdot \exists\, n_0 > 0 \cdot \forall\, n \geq n_0 \cdot f(n) < c\ g(n)$$

• $\omega(n)$:

Definition 5 *Seien* $f, g : \mathbb{N} \to \mathbb{R}_0^+$. *Dann nennt man* f von asymptotisch größerem Wachstum als g beim Parameterwachstum nach Unendlich, *wenn*

$$f(n) = \omega(g(n)) \underset{def}{\Leftrightarrow} \forall\, c > 0 \cdot \exists\, n_0 > 0 \cdot \forall\, n \geq n_0 \cdot c\ \ g(n) < f(n)$$

1.9 Redundanzen und Spezifikationen

1.9.1 Ein Beispiel

module *Homework*

definitions

types

39.0 *Student* is not yet defined;

40.0 $Stdset = Student\text{-set}$

41.0 state *Homework* of
.1 $Class : Stdset;$
.2 $Handed\text{-}in : Stdset;$
.3 $Not\text{-}handed\text{-}in : Stdset;$
.4 inv $mk\text{-}Homework(Class, Handed\text{-}in, Not\text{-}handed\text{-}in) \triangleq$
.5 $Handed\text{-}in \cup Not\text{-}handed\text{-}in = Class \wedge$
.6 $Handed\text{-}in \cap Not\text{-}handed\text{-}in = \{\}$
.7 init $mk\text{-}Homework(Class, Handed\text{-}in, Not\text{-}handed\text{-}in) \triangleq$
.8 $Not\text{-}handed\text{-}in = Class \wedge$
.9 $Handed\text{-}in = \{\}$
.10 end

operations

42.0 $Enrol\,(std : Student)$
.1 ext wr $Not\text{-}handed\text{-}in : Stdset$
.2 wr $Class : Stdset$
.3 pre $\neg\,(Std \in Class)$
.4 post $Not\text{-}handed\text{-}in = \overleftarrow{Not\text{-}handed\text{-}in} \cup \{std\} \wedge$
.5 $Class = \overleftarrow{Class} \cup \{std\}$
.6 errs $\mathrm{PRESENT} : (std \in Class) \rightarrow$
.7 $Not\text{-}handed\text{-}in = \overleftarrow{Not\text{-}handed\text{-}in} \wedge$
.8 $Class = \overleftarrow{Class}$

;

43.0 $Enquire\,(std : Student)\mathrm{erg}{:}\mathbb{B}$
.1 ext rd $Handed\text{-}in : Stdset$
.2 rd $Class : Stdset$

.3 pre $Std \in Class$

.4 post $erg = (std \in Handed\text{-}in)$

.5 errs NOT-IN-CLASS : $\neg\,(std \in Class) \rightarrow erg = false$

end *Homework*

Aufgabe : Spezifizieren Sie analog

```
Submit(std:Student) -- Student hat Hausarbeit
                    -- abgegeben

Remove(std:Student) -- Entferne Student aus
                    -- Klasse
```

Bemerkung 1.9.1

pre	VB
post	NB
errs	$Id_1 : B_1 \rightarrow B_1'$
	$\vdots$
	$Id_n : B_n \rightarrow B_n'$

ist äquiv. zu

pre
$VB \vee B_1 \vee \ldots B_n$
post
$(VB \wedge NB) \vee (B_1 \wedge B_1') \vee \ldots$

1.9.2 Implizite vs. explizite Operationsspezifikationen

Statt der abstrakteren impliziten Operationsspezifikation

```
44.0  OpName (p_1 : T_1, ... , p_n : T_n) r : T_r
 .1   ext wr ...
 .2   pre
 .3        ⋮
 .4   post
 .5        ⋮
 .6   errs
 .7        ⋮
```

ist es gerade bei der Programmentwicklung kurz vor der Codierung möglich, Operationen explizit zu definieren :

```
45.0  OpName : T_1 × ... × T_n --o--> T_r
 .1   OpName (p_1, ... , p_n)
 .2   ext wr ... △= statement
 .3   pre
 .4        ⋮
```

Ein Beispiel :

```
46.0  ADD-VALUE : Key × Value --o--> OVERWRITE | ADD
 .1   ADD-VALUE (key, value)
 .2   ext wr DATABASE : Key <--m--> Value △=
 .3     if key ∈ dom DATABASE
 .4     then (DATABASE (key) := value;
 .5           return OVERWRITE)
 .6     else (DATABASE := DATABASE ∪ {key ↦ value};
 .7           return ADD)
```

1.9.3 Implizite und explizite Funktionsspezifikationen

Man kann eine Funktion `max-int` folgendermaßen *explizit* spezifizieren:

$$
\begin{array}{ll}
47.0 & \mathit{max\text{-}int} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
.1 & \mathit{max\text{-}int}\,(i, j) \triangleq \\
.2 & \quad \mathsf{if}\ i \leq j\ \mathsf{then}\ j\ \mathsf{else}\ i \\
.3 & \mathsf{pre}\ \ \mathit{true}
\end{array}
$$

Die äquivalente *implizite* Darstellung lautet:

$$
\begin{array}{ll}
48.0 & \mathit{max\text{-}int}\,(i : \mathbb{Z}, j : \mathbb{Z}) r : \mathbb{Z} \\
.1 & \mathsf{pre}\ \ \mathit{true} \\
.2 & \mathsf{post}\ (r = i \vee r = j) \wedge i \leq r \wedge j \leq r
\end{array}
$$

1.9.4 Records und Invarianten

```
49.0  Koordinaten = compose Polar of
  .1                    Arg : ℝ
  .2                    Mod : ℝ
  .3                  end

  .4  inv mk-Polar(r, Θ) ≜ (r ≥ 0) ∧ (0 ≤ Θ) ∧ (Θ < 2π)
```

oder kürzer

```
50.0  Polar :: Arg : ℝ
  .1           Mod : ℝ

  .2  inv mk-Polar(r, Θ) ≜ (r ≥ 0) ∧ (0 ≤ Θ) ∧ (Θ < 2π)
```

Hier ein weiteres Beispiel:

```
51.0  Date :: Day : ℕ1
  .1          Month : ℕ1
  .2          Year : ℕ1

  .3  inv mk-Date(d, m, y) ≜
  .4      (1901 ≤ y) ∧ (y ≤ 2200) ∧
  .5      (m ≤ 12) ∧
  .6      d ≤ cases m :
  .7          9, 4, 6, 11 → 30,
  .8          2 → if ((y rem 4 = 0) ∧
  .9                  (y rem 100 ≠ 0)) ∨
  .10                 (y rem 400 = 0) then 29 else 28,
  .11         others → 31
  .12     end
```

Auch möglich sind Invarianten bei:

types

52.0 $Gerade = \mathbb{N}$

.1 inv $mk\text{-}Gerade(g) \triangleq$
.2 $\exists\, i : \mathbb{N} \cdot 2i = g$

oder

types

53.0 $Gerade = \{n : \mathbb{N} \cdot \exists\, i \in \mathbb{N} \cdot 2i = n\}$

Sei

$$\text{Heute} := mk\text{-}Date(21, 11, 1996)$$

Dann hat

$$\mu(Heute, Day \rightarrow Heute.Day + 1)$$

den Wert

$$mk\text{-}Date(22, 11, 1996)$$

1.9.5 Tupel

(Expliziter Verzicht auf Namen für Komponenten)

```
types
      Ebene = ℝ × ℝ;

dcl
      e : Ebene, f : Ebene;

e := mk-(0.0, 0.0);
...
let mk-(e1,e2) = e in
       f := mk-(e2,e1);
...
```

→ äquivalent zu $Ebene = \{(a, b) \mid a, b \in \mathbb{R}\ \}$

Index

Literaturverzeichnis

[Aho92] Alfred Aho. *Data structures and algorithms.* Addison-Wesley, 1992.

[Dij76] Edsger Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[Eng93] Herrmann Engesser. *Duden, Informatik.* Dudenverlag Mannheim, 1993.

[Jon90] Cliff Jones. *Systematic software development using VDM.* Prentice-Hall, 1990.

[Jun90] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI-Wissenschafts-Verlag, 1990.

[Knu72] Donald Knuth. *The Art of Computer Programming.* Addison-Wesley, 1972.

[Lin90] Charles Lins. *The Modula-2 software component library.* Springer, 1990.

[Mey90] Bertrand Meyer. *Object oriented software construction.* Prentice-Hall, 1990.

[Str77] Horst Strunz. *Entscheidungstabellentechnik.* Hanser, 1977.

[Tur94] John Turner. *The construction of formal specifications.* McGraw-Hill, 1994.

[Wir95] Niklaus Wirth. *Algorithmen und Datenstrukturen.* Teubner, 1995.