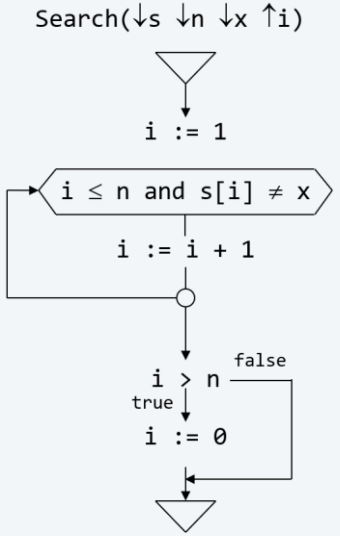
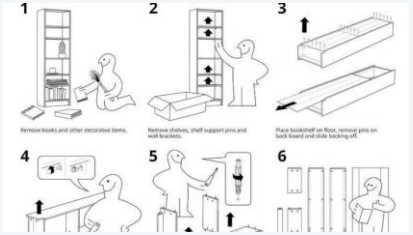


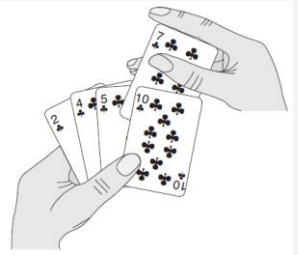
1 Der Algorithmusbegriff

1.1 Was ist ein Algorithmus?

Der Algorithmenbegriff ist ein zentraler Begriff der Informatik, aber Algorithmen sind keine Besonderheit der Informatik

Wir begegnen Algorithmen ständig im Alltag, oft ohne sie als solche zu erkennen





"Algorithmisches Denken ist eine allgemeine Methode, um Wissen zu organisieren" (D. Knuth)

"In nur wenigen Jahrzehnten hat das Computational Thinking, also das analytische, von Algorithmen geprägte Denken unser Leben, Arbeiten und Spielen von Grund auf verändert." (P. Curzon, P.W. McOwan)

Was ist ein Algorithmus?

Algorithmus = Problemlösungsvorschrift

- die Lösung einer Aufgabe bedarf einer eindeutigen Vorschrift, die genau festlegt, welche Aktionen (Handlungen) nacheinander auszuführen sind
- die formale Beschreibung einer solchen Lösungsvorschrift nennen wir Algorithmus (Algorithmus beschreibt Handlungen und deren Reihenfolge)

Ein Algorithmus ist im Wesentlichen eine „Anleitung zur Lösung einer Aufgabenstellung“

Beispiel: Größter gemeinsamer Teiler nach Euklid

Wenn p und q zwei positive, ganze Zahlen sind, lässt sich der größte gemeinsame Teiler (gcd) von p und q berechnen:

Schritt 1: Dividiere p durch q und bilde Rest r der Division

Schritt 2: Wenn $r = 0$, dann ist q der gesuchte gcd
Wenn $r \neq 0$, dann nenne q in p, und r in q um und wiederhole Schritt 1 und Schritt 2 solange, bis $r = 0$ geworden ist

Ablauf für
p = 378 und q = 216

validieren: gehts auch für q größer p?
Sei p=378 und q=216
0 Rest, es dreht sich um und geht bei der Tabelle rechts weiter

Schritt	p	q	r
1	378	216	162
2	216	162	162
1	216	162	54
2	162	54	54
1	162	54	0
2	162	54	0

gcd(378, 216) = 54

Eigenschaften von Algorithmen

Aus dem Beispiel lassen sich folgende Eigenschaften von Algorithmen ableiten

- Aneinanderreihung mehr oder weniger komplexer Aktionen
- Aktionen setzen sich i.d.R. aus Elementaraktionen zusammen
  z.B. p/q - nicht aufteilbare Aktion
- Vorhandensein von Datenobjekten, auf die Aktionen angewandt werden und deren Zustandsänderung das Ergebnis einer Aktion darstellt
- Jedes Datenobjekt gehört zu einem bestimmten Wertebereich
  Beispiel: positive, ganze Zahlen
- Es gibt einen Ausführenden, für den der Algorithmus bestimmt ist, wir nennen diesen Prozessor

Definition (*Prozessor*):
Unter einem *Prozessor* verstehen wir die treibende Kraft, die die Aktionen eines Algorithmus ausführt, z.B. Mensch oder Maschine. Der Algorithmus muss für den Prozessor eindeutig interpretierbar und ausführbar sein.

Beispiel Wertebereich: Funktioniert p=42, q=-12?
-- -12 liegt nicht im Wertebereich, ist also nicht relevant, Algorithmus ist also nicht anwendbar

Eigenschaften und Begriffsdefinition

Ein Algorithmus ist ein Problemlösungsverfahren mit folgenden charakteristischen Merkmalen

- ermittelt gesuchte aus gegebenen Datenobjekten (Zahlen, Texte, Bilder etc.)
- wird schrittweise ausgeführt
- ist (für den Prozessor, für den es bestimmt ist) eindeutig interpretierbar und ausführbar

Definition (*Algorithmus*):
Ein Algorithmus ist eine vollständige, präzise und in einer Notation oder Sprache mit exakter Definition abgefasste, endliche Beschreibung eines schrittweisen Problemlösungsverfahrens zur Ermittlung gesuchter Datenobjekte (ihrer Werte) aus gegebenen Werten von Datenobjekten, in dem jeder Schritt aus einer Anzahl ausführbarer, eindeutiger Aktionen und einer Angabe über den nächsten Schritt besteht.

Eigenschaften

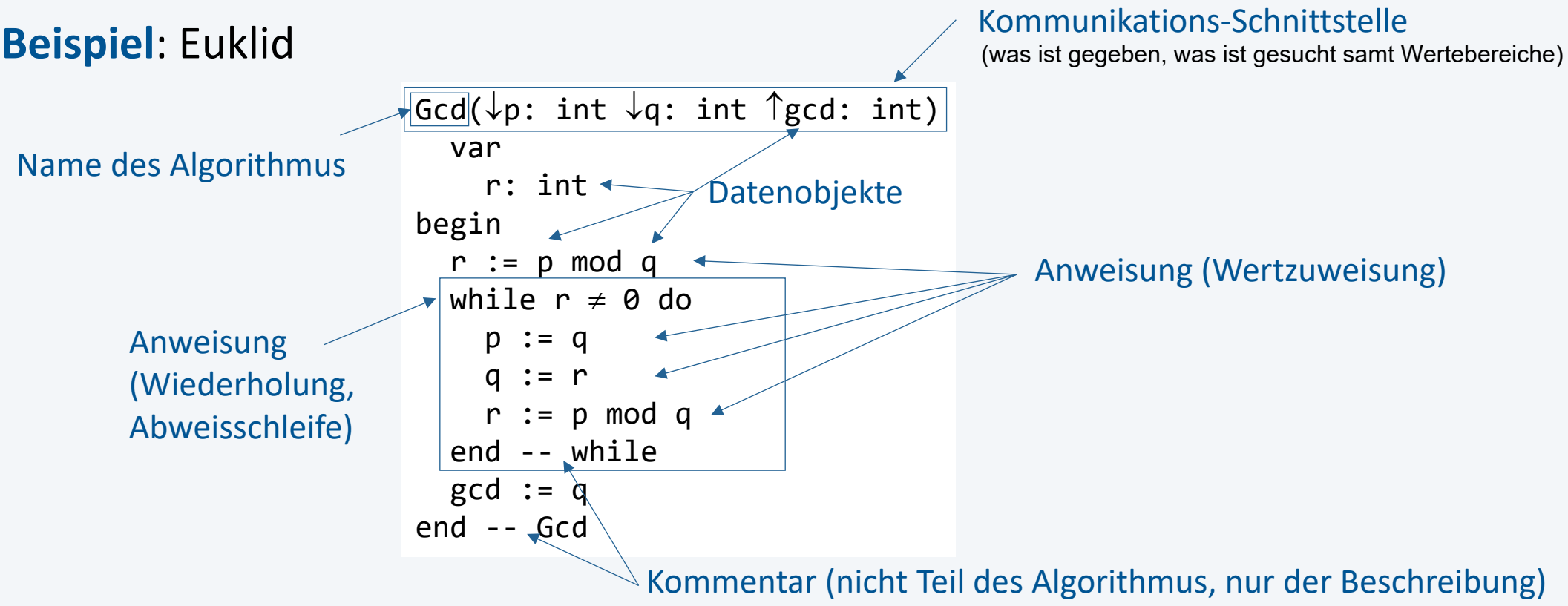
Korrektheit	Algorithmus erfüllt die seiner Entwicklung zugrundeliegende Spezifikation (siehe Kapitel 2)
Vollständigkeit	Der Algorithmus muss vollständig sein
Eindeutigkeit der Aktionen	Keine Interpretationsmöglichkeit des Ausführenden (Prozessor) Keine Unklarheiten bezüglich der Ausführung eines Schritts und der Wahl des nächsten Schritts
Ausführbarkeit der Aktionen	Vom Ausführenden darf nichts Unmögliches verlangt werden Der Algorithmus darf nicht in einen undefinierten Zustand geraten, z.B. bei Division durch 0
Statische Endlichkeit	Algorithmus muss mit endlich vielen Zeichen beschrieben werden können
Dynamische Endlichkeit	Ein Algorithmus, der die Werte für die gesuchten Datenobjekte oder gewünschte Effekte nach endlich vielen Schritten liefert, heißt abbrechend oder terminierend
Effizienz	Algorithmus erfüllt seinen Zweck unter bestmöglicher Ausnutzung aller benötigten Ressourcen (Ausführungszeit, Speicherplatz, Energie)

1.2 Aufbau von Algorithmen

Algorithmen bestehen aus Datenobjekten und Anweisungen (Aktionen) zu deren Manipulation

Algorithmen kommunizieren über eine Schnittstelle mit anderen Algorithmen

Beispiel: Euklid




```
Gcd(↓p: int ↓q: int ↑gcd: int)
  var
    r: int
begin
  r := p mod q
  while r ≠ 0 do
    p := q
    q := r
    r := p mod q
  end -- while
  gcd := q
end -- Gcd
```

Datenobjekte und Operationen

Datenobjekte können Werte aus verschiedenen Wertebereichen annehmen

`42`	`13`	`-7`	… ganze Zahlen (zum Zählen, Index)
`17.3`	`-27.3`	`0.07`	… reelle Zahlen (Berechnungen aus der Natur)
`'A'`	`'Z'`	`'4'`	… Zeichen aus Zeichensatz (Kommunikation Maschine - Mensch)
`true`	`false`		… Wahrheitswerte (Überprüfung von Werten)

Die möglichen Operationen hängen (auch) vom Wertebereich ab

eindeutig		kind of definiert - programmiersprachenabhängig, deshalb wird das eher vermieden.	
`13 * 14 + 5`	✓	`13 * 'A'`	✗
`3.14 * -1.0 + 9.0`	✓	`0 < true`	✗
`'A' < 'Z'`	✓	`42 * false`	✗

Algos werden so geschrieben, dass sie möglichst universell einsetzbar sind

Wertebereich und erlaubte Operationen werden durch Typisierung der Datenobjekte festgelegt

Wertzuweisung

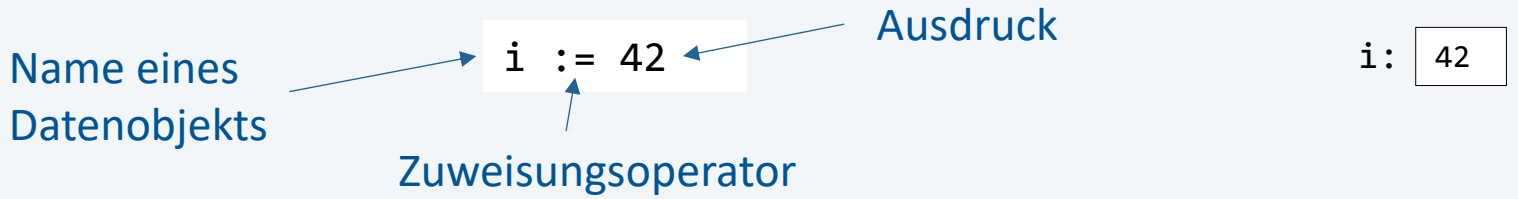
Ausdruck: auswertbar, hat einen Wert

```
Name := Ausdruck
```

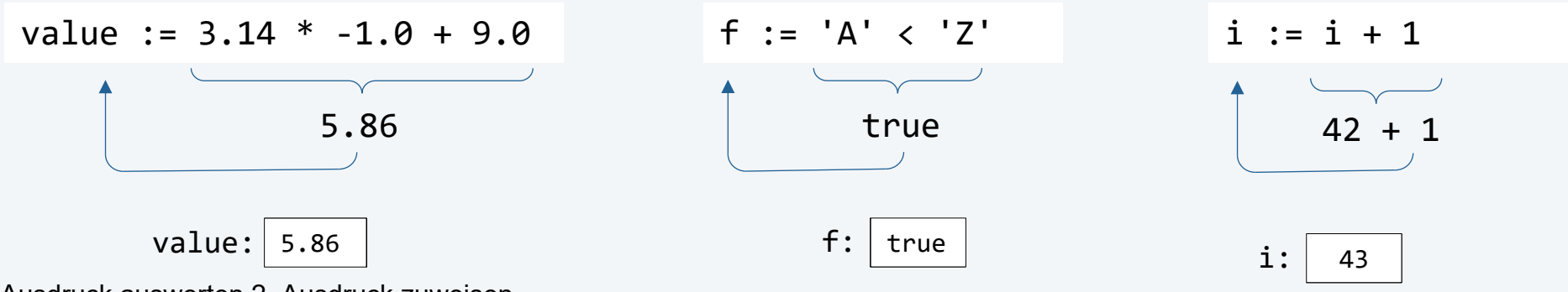
Der Wert eines Datenobjekts wird durch Wertzuweisung verändert

Wertzuweisung erfolgt durch Zuweisungsoperator (`:=`) nach Auswertung des Ausdrucks auf der rechten Seite

Beispiel: dem Datenobjekt `i` wird der Wert 42 zugewiesen



Auf der rechten Seite können beliebige Ausdrücke vorkommen



1. Ausdruck auswerten 2. Ausdruck zuweisen

Wertzuweisung

Werden exakt in der Reihenfolge ausgeführt, in der sie aufgeschrieben sind

```
p := q
q := r
```

↓ Ausführungsreihenfolge

Beispiel: Veränderung der Reihenfolge

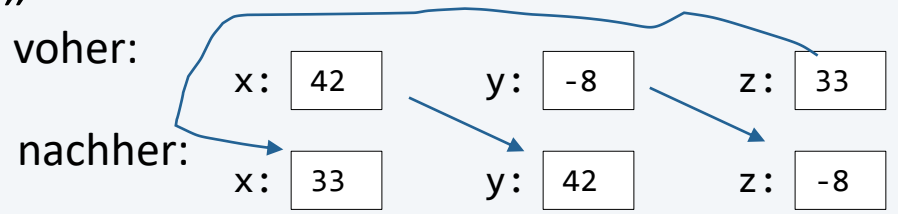
```
i := 3
j := i * i
i := i + 1
```

```
i := 3
i := i + 1
j := i * i
```

Beispiel: Inhalt von zwei kompatiblen Datenobjekten a und b vertauschen

```
t := b
b := a
a := t
```

Aufgabe: Inhalt in Datenobjekten x, y, z „nach rechts rotieren“



1.3 Datenobjekte und ihre Datentypen

Wir unterscheiden grundsätzlich zwei Arten von Datenobjekten

- solche, deren Wert während der Ausführung eines Algorithmus verändert werden kann, wir nennen sie deshalb Variable
- solche, deren Wert nicht verändert werden kann, wir nennen sie deshalb Konstante

Zu jedem Datenobjekt gehört ein bestimmter Datentyp

Definition (*Datentyp*):
Der Datentyp definiert die Wertemenge, aus der eine Variable einen Wert annehmen kann oder aus der eine Konstante stammt und die Operationen, die auf den entsprechenden Datenobjekten erlaubt sind.

Wir unterscheiden zwischen elementaren und strukturierten Datentypen und -objekten

Elementare Datenobjekte und Datentypen

Elementare Datentypen (kurz Typen) spezifizieren

- den Wertebereich der Datenobjekte
- die Operationen, die auf den Datenobjekte ausgeführt werden können
- die Semantik der Operationen

Das Ergebnis einer Operation hängt vom Typ der Operanden ab,
z.B. gilt in Pascal:

- ganze Zahl + ganze Zahl $\Rightarrow$ ganze Zahl `17 + 4` $\Rightarrow$ `21`
- reelle Zahl + reelle Zahl $\Rightarrow$ reelle Zahl `17.0 + 4.0` $\Rightarrow$ `21.0`
- ganze Zahl + reelle Zahl $\Rightarrow$ reelle Zahl `17 + 4.3` $\Rightarrow$ `21.3`

Die wichtigsten elementaren Datentypen für unstrukturierte Datenobjekte sind:

- `int`, `real`, `char`, `bool`

Deklaration von Datenobjekten

Wir legen fest, dass jedes Datenobjekt (Variable oder Konstante) vor seiner Verwendung deklariert werden muss.

Wir führen dazu die Sprachkonstrukte `var` und `const` ein und definieren Datenobjekte folgendermaßen (Beispiele):

```
var
  temperature: real
  startNumber: int
  isPrime: bool
```

```
const
  pi: real = 3.141592
  maxSize: int = 100
```

Datenobjekte in diesem Sinne haben einen Namen, mit dem sie identifiziert werden können (z.B. `pi`), einen Datentyp (z.B. `real`) und einen Wert (z.B. `3.141592`).

Der Wert ist für Variable nach der Deklaration undefiniert; für Konstante wird er bereits bei der Deklaration festgelegt.

Der Datentyp *Integer* (kurz `int`)

- umfasst Teilmenge der ganzen Zahlen

  ..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...

- Wertebereich hängt von der Anzahl der verwendeten Bits ab
- Wert einer 16-Bit-Zahl $b_{15}b_{14}b_{13}b_{12}\cdots b_3b_2b_1b_0$ ist

$$b_{15}\cdot 2^{15} + b_{14}\cdot 2^{14} + \cdots + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0$$

- mit n Bits kann man 2^n verschiedene Zahlen darstellen

Variante 1: Ein Bit für Vorzeichen

- um positive und negative Zahlen darstellen zu können, sind negative Zahlen durch b_{15} = 1 gekennzeichnet, bleiben $b_{14}\cdots b_0$ für die verschiedenen Werte
- ergibt Wertebereich für 16-Bit-Zahlen: -32767 .. 32767

2^0	1
2^1	2
2^2	4
2^3	8
	16
	32
	64
	128
2^8	256
	512
	1024
	2048
	4096
	8192
	16384
2^{15}	32768
2^{16}	65536

Zahlendarstellung

Variante 2: Mit 2-Komplement

- um einfach mit positiven und negativen Zahlen rechnen zu können, werden negative Zahlen als so genannte Komplemente dargestellt

$$-x = 2^{16} - x, \text{ für } n = 16$$

- Zahlenbeispiele für $n = 16$

0000 0000 0000 0000	0	0
0000 0000 0000 0001	2^0	1
0111 1111 1111 1111	$2^{15} - 1$	32767

1111 1111 1111 1111	$2^{16} - 1$	-1
1111 1111 1111 1110	$2^{16} - 2$	-2
1000 0000 0000 0000	$2^{16} - 2^{15}$	-32768

größte Zahl

kleinste Zahl

- Rechenbeispiele für $n = 4$

0011 + 1111 = 0010, $3 + (-1) = 2$

0010 + 1101 = 1111, $2 + (-3) = -1$

Operationen

Operationen für einen Operanden mit Ergebnis vom Typ `int`

`+`	positives Vorzeichen
`-`	negatives Vorzeichen

Operationen für je zwei Operanden mit Ergebnis vom Typ `int`

`+`	für die Addition
`-`	für die Subtraktion
`*`	für die Multiplikation
`div`	für die ganzzahlige Division (Rest wird nicht berücksichtigt)
`mod`	für den Rest bei der ganzzahligen Division

Operationen für je zwei Operanden mit Ergebnis vom Typ `bool`

`=`	ist gleich
`<`	ist kleiner als
≤	ist nicht größer als

≠	ist ungleich
`>`	ist größer als
≥	ist nicht kleiner als

Typische Ausdrücke mit `int`-Datenobjekten

Beispiele:

Ausdruck	Wert	Anmerkung
`+99`	`99`	positives Vorzeichen
`-99`	`-99`	negatives Vorzeichen
`5 + 3`	`8`	Addition
`5 - 3`	`2`	Subtraktion
`5 * 3`	`15`	Multiplikation
`5 div 3`	`1`	ganzzahlige Division
`5 mod 3`	`2`	Rest bei der ganzzahligen Division
`1 div 0`		Laufzeitfehler
`3 * 5 - 2`	`13`	`*` hat Vorrang
`3 + 5 div 2`	`5`	`div` hat Vorrang
`3 - 5 - 2`	`-4`	von links nach rechts
`(3 - 5) - 2`	`-4`	mit Klammerung (besser)

Der Datentyp *Real* (`real`)

Mit Gleitkommazahlen können sehr kleine und sehr große Werte repräsentiert werden

- z.B. Ladung eines Elektrons, ca. $1.602 \cdot 10^{-19}$ Coulomb
- z.B. ein Lichtjahr, ca. $9.46 \cdot 10^{15}$ Meter

intern dargestellt als Mantisse und Exponent

Wertebereich und Genauigkeit hängen von der Aufteilung der Bits für Mantisse und Exponent ab

Operationen: +, -, *, /, =, ≠, <, >, ≤, ≥

Beispiel: $pi = 0.3141592 \cdot 10^1$
Mantisse: 3141592,
Exponent: 1

Mantisse sind immer die Kommazahlen bei Verschiebung auf 0, ; der Exponent erzeugt wieder den richtigen Wert

Ausdruck	Wert
`3.141 + 0.03`	`3.171`
`5.0 / 3.0`	`1.6666666666666667`

Konsequenz: solche Rundungsfehler können sich summieren. → Also darf man nicht real Zahlen auf Gleichheit prüfen! → Eine Lösung wär zu prüfen, ob die Differenz der Zahlen innerhalb eines definierten Toleranzbereiches liet

Achtung: durch Rundungsfehler kann Vergleich mit = oder ≠ unerwünschtes Ergebnis liefern

man kann nur Zahlen mit endlicher Genauigkeit speichern

real ist aus dem Grund z.B. nicht für Geld geeignet. Man kann beispielsweise die umgerechneten Centbeträge als int speichern

Der Datentyp *Boolean* (kurz `bool`)

- umfasst Wahrheitswerte `true` und `false` (es gilt `false` < `true`)
- Operationen für je zwei Operanden mit Ergebnis vom Typ `bool`

`and`	Konjunktion

`or`	Disjunktion

- Operation für einen Operand mit Ergebnis vom Typ `bool`

`not`	Negation

- Vergleichsoperationen =, ≠, <, >, ≤, ≥

a	**b**	**a and b**	**a or b**
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

a	**not a**
`false`	`true`
`true`	`false`

Typische Ausdrücke mit `bool`-Ergebnissen

Aufgabe: Erstellen Sie einen Ausdruck, der überprüft, ob die Werte in den drei `int`-Datenobjekten a, b und c gleich sind.

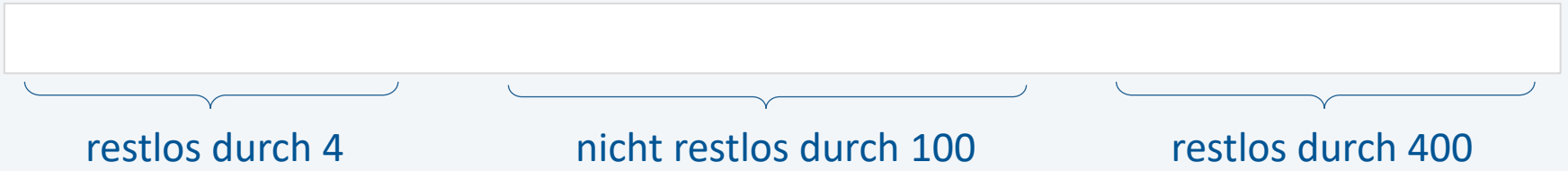
Aufgabe: Gegeben ist ein `int`-Datenobjekt `year`, das ein vierstelliges Jahr repräsentiert, z.B. 2024. Erstellen Sie einen Ausdruck, der überprüft, ob das Jahr in `year` ein Schaltjahr ist.

`(year mod 4) = 0`

`(year mod 100) = 0`

`(year mod 400) = 0`

Schaltjahr:
1. Ein Jahr ist ein Schaltjahr, wenn es **restlos durch 4** teilbar ist.
2. Jedoch ist ein Jahr **kein** Schaltjahr, wenn es **restlos durch 100** teilbar ist.
3. Ein Jahr ist jedoch ein Schaltjahr, wenn es **restlos durch 400** teilbar ist.



Der Datentyp *Character* (kurz `char`)

Anwendung:
x>=a und x<=z --> x ist ein Kleinbuchstabe! z.B.
Prüfen von Passwortrequirements

- Repräsentiert die Menge eines Zeichensatzes (i.d.R. der verfügbaren druckbaren Zeichen), für uns mindestens

```
'a', 'b', … 'z'
```

```
'A', 'B', … 'Z'
```

```
'0', '1', … '9'
```

- Teilmengen der Großbuchstaben, Kleinbuchstaben und Ziffern sind geordnet und zusammenhängend

```
'a' < 'b' … < 'z'
```

```
'A' < 'B' … < 'Z'
```

```
'0' < '1' … < '9'
```

- Operationen =, ≠, <, ... für je zwei Operanden vom Typ `char` mit Ergebnis vom Typ `bool`
- Typische Ausdrücke mit `char`-Datenobjekten

```
var
  x: char
```

```
'a' ≤ x and x ≤ 'z'
```

`true` … x ist Kleinbuchstabe

```
'A' ≤ x and x ≤ 'Z'
```

`true` … x ist Großbuchstabe

```
'0' ≤ x and x ≤ '9'
```

`true` … x ist Ziffer

Ordinalzahlen

ASCII Zeichensatz hat Symbole und Buchstaben durchnummeriert. Diese Nummer = Ordinalzahl. int("c") liefert den Wert dieser Tabelle, z.B. 48. Das geht in beide Richtungen: char(48) = "c"

- Aufgrund der Ordnung kann jedem Zeichen eine Ordinalzahl (vom Typ `int`) zugeordnet werden

```
var
  c: char
  i: int
```

```
Int(↓c)
```
… liefert Ordinalzahl für Zeichen `c`

```
Char(↓i)
```
… liefert Zeichen der Ordinalzahl `i`

- es gilt

```
Int(↓Char(↓i)) = i
```
wenn `Char(↓i)` definiert ist

```
Char(↓Int(↓c)) = c
```

Im Speicher wird die Ordinalzahl gespeichert. Interpretiert als char, wird der zugehörige Buchstabe ausgegeben

- da Teilmengen zusammenhängend sind gilt

```
Int(↓c) - Int(↓'0')
```
Position von `c` innerhalb der Ziffern

```
Char(↓i + Int(↓'0'))
```
liefert `i`-te Ziffer

```
c := "7"          //man möchte den Wert als int in eine andere Variable speichern
i := c    //geht nicht, weil unterschiedliche Datentypen
int(c)-int("0")       // dadurch dass die Ordinalzahlen subtrahiert werden, entsteht der tatsächliche int
```

Der Datentyp *String* (`string`)

Eine Zeichenkette (`string`) ist eine Folge von Zeichen, deren Anzahl von Zeichen begrenzt ist (maximale Länge); die tatsächliche Anzahl der Zeichen nennen wir „aktuelle Länge“

```
var
  name: string
  city: string
```

maximale Länge
z.B. 255 Zeichen

Durch Wertzuweisung kann der Inhalt (Wert) und die aktuelle Länge einer Stringvariablen verändert werden

```
name := "FH"
city := "Hagenberg"
```

aktuelle Länge 2
aktuelle Länge 9

Die Funktion `Length(↓name)` liefert die aktuelle Länge (Zeichenanzahl) der Stringvariable `name`

```
Length(↓name)
Length(↓city)
```

liefert 2
liefert 9

length ist eine
Operation auf strings

Operationen

Konkatenation mit Operator +

```
var
  s: string
  len: int
```

```
s := name + " " + city
len := Length(↓s)
```

liefert "FH Hagenberg"

liefert 12

Datentypen: bei s[2] := 'F'

s ... string
'F' .... char
s[2] .... char .... liefert sicher nur einen Buchstaben

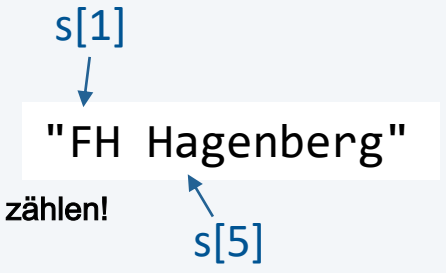
Zugriff auf einzelne Zeichen

```
var
  c: char
```

```
c := s[5]
```

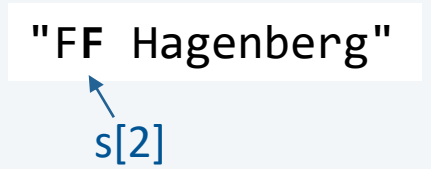
oft beginnt man bei 0 zu zählen!

liefert 'a'




```
s[2] := 'F'
```

ändert 'H' auf 'F'



der Zugriff ist nur auf Zeichen im Bereich `1..Length(↓s)` möglich

-- mögliche Programmierfehler

Beispiele

Beispiel: Hundealter in Menschenalter umrechnen

```
var
  dogYears, humanYears: real
```

```
humanYears := 16.0 * Ln(↓dogYears) + 31.0
```

Beispiel: Lösung von $x^2 + bx + c = 0$

```
var
  b, c, sq, x1, x2,
  discriminant: real
```

```
discriminant := b*b – 4.0 * c
sq := Sqrt(↓discriminant)
x1 := (-b + sq) / 2.0
x2 := (-b - sq) / 2.0
```

vll vorher Diskriminante auf Positivität oder 0 prüfen, sonst sqrt(-x) --> undefiniert in R

Beispiel: Validierung einer Zellreferenz, z.B 'A42'

```
var
  s: string
  correctInput: bool
```

```
correctInput := Length(↓s) = 3
                and 'A' ≤ s[1] and s[1] ≤ 'Z'
                and '0' ≤ s[2] and s[2] ≤ '9'
                and '0' ≤ s[3] and s[3] ≤ '9'
```

1.4 Verzweigungen und Schleifen

Verzweigung

```
if Bedingung then
  Anweisungen
end
```

- wenn Bedingung erfüllt ist, werden Anweisungen zwischen `then` und `end` ausgeführt, sonst werden diese Anweisungen ignoriert
- Bedingungsauswertung muss Wahrheitswert {`true`, `false`} liefern
- beliebig viele Anweisungen zwischen `then` und `end`

Beispiele

```
if x < 0 then
  x := -x
end
Write(↓x)
```

```
if a > b then
  t := a
  a := b
  b := t
end
```

```
if a > b then
  if a > c then
    Write(↓'a')
  end
end
```

```
if a < b then
  min := a
end
if a >= b then
  min := b
end
```

umständlich!
siehe nächste Seite

Verzweigung

```
if Bedingung then
  Anweisungen1
else
  Anweisungen2
end
```

- wenn Bedingung erfüllt ist, werden $Anweisungen_1$ zwischen `then` und `else` ausgeführt
- sonst werden $Anweisungen_2$ zwischen `else` und `end` ausgeführt
- es werden immer entweder die $Anweisungen_1$ oder die $Anweisungen_2$ ausgeführt

Beispiele

```
if a < b then
  min := a
else
  min := b
end
```

```
if a < b then
  min := a
  max := b
else
  min := b
  max := a
end
```

```
if age >= 16 then
  Write(↓'watch movie')
else
  if age >= 12 and withParents then
    Write(↓'watch movie with parents')
  else
    Write(↓'not allowed to watch movie')
  end
end
```

Mehrwegverzweigung (Fallunterscheidung)

```
case Ausdruck of
  Fall_1: Anweisungen_1
  Fall_2: Anweisungen_2
  ...
  Fall_n: Anweisungen_n
  otherwise: Anweisungen
end
```

- Es wird der Anweisungsblock ausgeführt, dessen Fallangabe mit dem Wert des Ausdrucks übereinstimmt
- Die Fallangaben dürfen sich nicht überschneiden

Beispiel

```
case grade of
  1: Write(↓'Sehr gut')
  2: Write(↓'Gut gut')
  3: Write(↓'Befriedigend')
  4: Write(↓'Genügend')
  5: Write(↓'Nicht genügend')
  otherwise: Write(↓'Fehler: Ungültige Note')
end -- case
```

```
if grade = 1 then
  Write(↓'Sehr gut')
else
  if grade = 2 then
    Write(↓'Gut gut')
  else
    if grade = 3 then
      Write(↓ Befriedigend')
    else
      if grade = 4 then
        Write(↓'Genügend')
      else
        if grade = 5 then
          Write(↓'Nicht genügend')
        else
          Write(↓'Fehler: Ungültige Note')
        end
      end
    end
  end
end
```

Abweisschleife (while-Schleife)

```
while Bedingung do
  Anweisungen
end
```

- solange die Bedingung erfüllt ist, werden die Anweisungen zwischen `do` und `end` (Schleifenrumpf) ausgeführt
- Bedingungsauswertung muss Wahrheitswert {`true`, `false`} liefern
- beliebig viele Anweisungen zwischen `do` und `end`

Beispiel: Berechne die Summe 1 + 2 + 3 + … + n

```
sum := 0
i := 1
while i ≤ n do
  sum := sum + i
  i := i + 1
end
```

Beachte: Schleifenrumpf muss nicht notwendigerweise ausgeführt werden (deshalb die Bezeichnung Abweisschleife)

Abweisschleife (while-Schleife)

Beispiel: Quadrate einer Zahlenfolge berechnen und ausgeben

```
Read(↑value)
while value > 0 do
  sq := value * value
  Write(↓'square = ' ↓sq)
  Read(↑value)
end
```

Eingabe:
  3 5 9 0
Ausgabe:
  square = 9
  square = 25
  square = 81

Eingabe:
  25 0
Ausgabe:
  square = 625

Eingabe:
  0
Ausgabe:

Beispiel: Vielfache durch ‚x' ersetzen

```
Read(↑x)
if x > 0 then
  i := 1
  while i <= 100 do
    if i mod x = 0 then
      Write(↓'x ')
    else
      Write(↓i ↓' ')
    end -- while
    i := i + 1
  end -- while
end -- if
```

Eingabe:
  7
Ausgabe:
  1 2 3 4 5 6 x 8 9 10 11 12 13 x 15 16 17 18 19 20 x …

Eingabe:
  2
Ausgabe:
  1 x 3 x 5 x 7 x 9 x 11 x 13 x 15 x 17 x 19 x 21 x …

Durchlaufschleife (repeat-Schleife)

```
repeat
  Anweisungen
until Bedingung
```

Anweisungen werden min. 1 Mal ausgeführt, weil die Bedingung erst am Ende geprüft wird

läuft bis die Bedingung erfüllt wird - umgekehrt zur while Schleife, welche läuft, solange die Bedingung erfüllt ist.

- Anweisungen werden solange ausgeführt, bis Bedingung erfüllt ist
- Bedingung ist Ausdruck, der Wahrheitswert liefert
- Beliebig viele Anweisungen zwischen `repeat` und `until`

Beispiel: Berechne Summe 1 + 2 + 3 + … + n

```
sum := 0
i := 1
repeat
  sum := sum + i
  i := i + 1
until i > n
```

zunächst werden Anweisungen ausgeführt

danach wird Ausdruck berechnet

Bedingung genau umgekehrt zur while Schleife

Wann verwendet man was: Wenn mindestens einmal ausgeführt wird, bildet sich die repeat-Schleife an.

Zählschleife

```
for Name := Startwert to Endwert do
  Anweisungen
end
```

```
for Name := Startwert downto Endwert do
  Anweisungen
end
```

- Anweisungen werden für alle Werte von Name (z.B. `i`) von Startwert bis Endwert ausgeführt
- die Laufvariable (z.B. `i`) ist ganzzahlig und läuft in äquidistanten Schritten aufwärts oder abwärts

integer eignet sich

- Endwert ist vorher bekannt

(zum Zeitpunkt des Schleifeneingangs muss n bekannt sein. n soll nicht abhängig von Anweisungen in der Schleife)

- Laufvariable nicht in der Schleife verändern
- Wert der Laufvariable nach Schleife ist unbekannt →

in manchen Programmiersprachen hat die Laufvariable nach der Schleife einen bestimmten Wert. Da sich das je nach Programmiersprache unterscheidet, soll man bei späterer Verwendung der selben Variable den gewünschten Wert neu zuweisen --> guter Programmierstil

Beispiel: Berechne Summe 1 + 2 + 3 + … + n

```
sum := 0
for i := 1 to n do
  sum := sum + i
end
```

Laufvariable wird automatisch inkrementiert

1.5 Algorithmen und ihre Schnittstellen

Algorithmen kommunizieren i.d.R. mit ihrer „Umwelt“, d.h. mit anderen Algorithmen (von denen sie benutzt werden) und/oder mit Geräten (von denen sie benötigte Daten „lesen“ und/oder auf denen sie ihre Ergebnisse ausgeben)

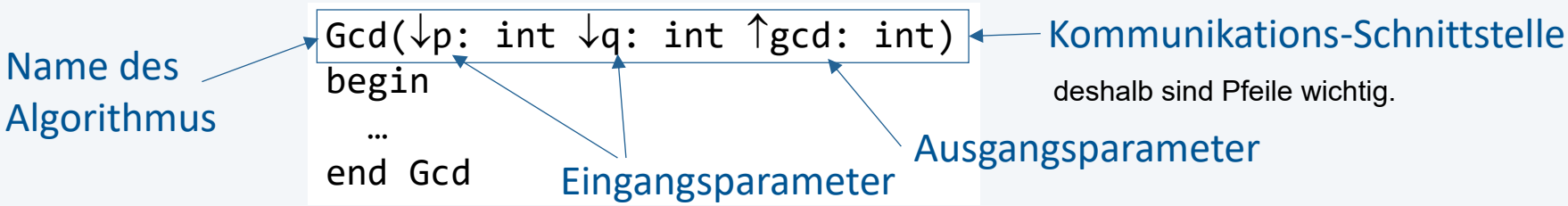
Zur Kommunikation mit der Umwelt besitzen Algorithmen sogenannte Schnittstellen, uzw.

- eine Kommunikations-Schnittstelle (für die Kommunikation mit anderen Algorithmen) und,
- wenn erforderlich, eine Geräteschnittstelle (für die Kommunikation mit Geräten)

Kommunikations-Schnittstelle - Deklaration

Dient zur Kommunikation mit anderen Algorithmen

Deklaration eines Algorithmus beschreibt, was der Algorithmus leistet

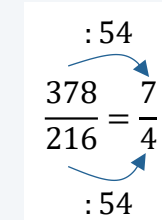


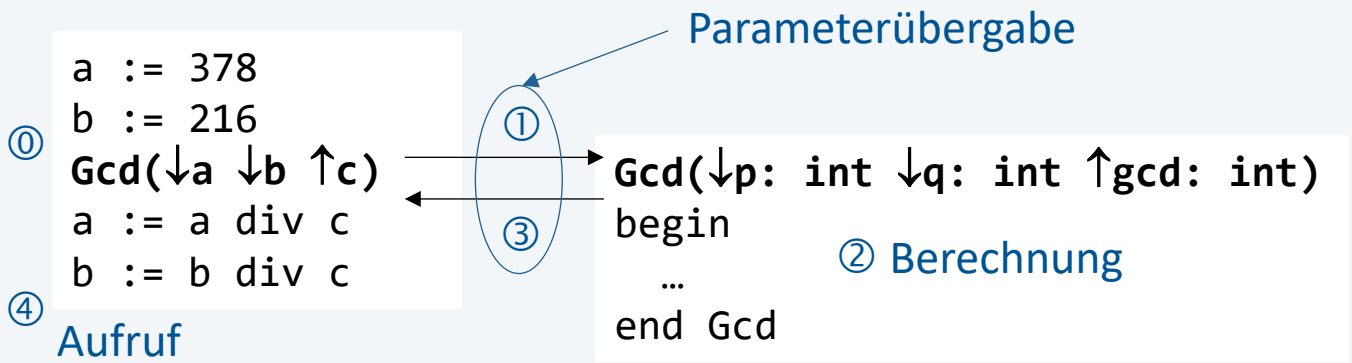
- Eingangsparameter (↓) liefern dem Algorithmus jene Werte, die er bei seiner Ausführung für seine Berechnungen benötigt
- Ausgangsparameter (↑) liefern dem Rufer am Ende der Algorithmen-ausführung die Ergebnisse der Berechnungen des Algorithmus
- Übergangsparameter (↓↑) liefern dem Algorithmus Werte, der Algorithmus ändert diese Werte und stellt sie nach der Algorithmenausführung dem Rufer zur Verfügung

Kommunikations-Schnittstelle - Aufruf

Verwendung (Aufruf) eines Algorithmus

Beispiel: Euklid





Ablauf:

Zustand	a	b	c	p	q	gcd
⓪	378	216				
①	378	216		378	216	
②	378	216		162	54	54
③	378	216	54	162	54	54
④	7	4	54			

Kommunikations-Schnittstelle - Aufruf

Weitere Beispiele

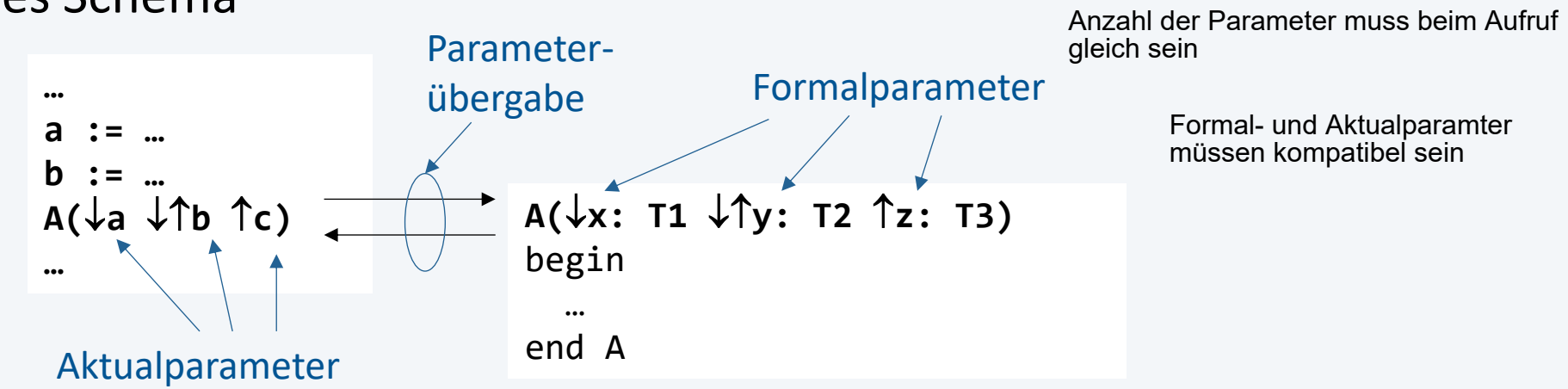
```
Minimum(↓a ↓b ↑min)
```

Berechne Minimum von a und b

```
Swap(↓↑a ↓↑b)
```

Vertausche Werte von a und b

Allgemeines Schema



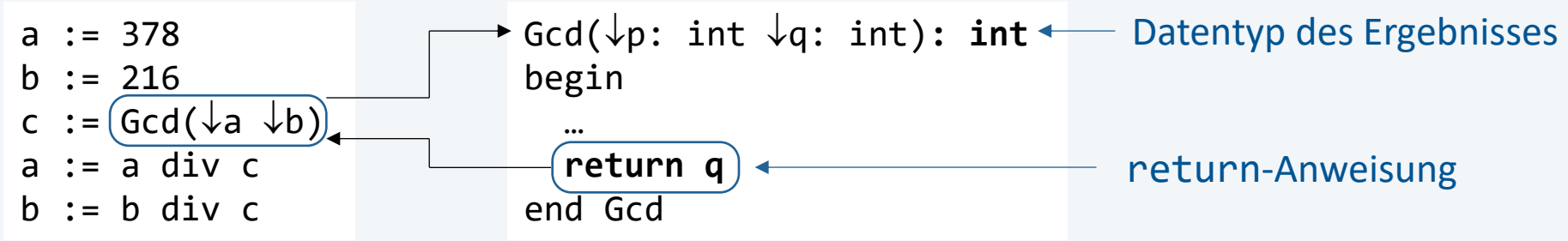
- Anzahl und Typen der Formal- und Aktualparam. müssen übereinstimmen
- empfohlene Reihenfolge: Eingangs-, Übergangs- und Ausgangsparameter

--> beim Definieren

Funktionsalgorithmen

Ein Algorithmus, der genau einen Wert liefert, kann auch als Funktion im Sinne der Mathematik, z.B. $\sin x$, aufgefasst werden

- Der Ausgangsparameter entfällt und wird durch den Datentyp des Ergebnisses (nach der Parameterliste) ersetzt
- Die Anweisung `return` *expr* liefert den Wert des Ausdrucks *expr* an den rufenden Algorithmus



- Aufruf kann auch in einem Ausdruck erfolgen

```
y := Cos(↓x) / Sin(↓x)
```

einfacher als

```
Cos(↓x ↑cx)
Sin(↓x ↑sx)
y := cx / sx
```

Vorteil: ein Funktionsaufruf ist ein direkter Ausdruck und es kann damit gerechnet werden z.B. gcd(a,b)+5

Return-Anweisung (`return`)

`return`

- Ermöglicht das vorzeitige Beenden einer Algorithmenausführung (z.B. wenn das Ergebnis bereits feststeht oder eine Fortsetzung der Ausführung sinnlos ist)

Beispiel

```
FindValue(↓s: string ↓x: char ↑i: int)
begin
  i := Length(↓s)
  while i > 0 do
    if s[i] = x then
      return
    end -- if
    i := i - 1
  end -- while
end FindValue
```

suchen von hinten nach vorne

Wenn der Buchstabe nicht enthalten ist, wird der Wert 0 zurückgegeben, weil die Schleife bei i=0 beendet wird

Return-Anweisung und Funktions-Algorithmus

```
return Ausdruck
```

- Wenn in der Return-Anweisung ein Ausdruck angeführt wird, dann wird der Wert des Ausdruckes unter dem Namen des Algorithmus (der die Return- Anweisung enthält) an den rufenden Algorithmus übergeben.
- Dem Namen des Algorithmus muss (in der Schnittstellendefinition) ein Datentyp zugeordnet werden (Sicherstellung der Typkompatibilität!)

Beispiel

```
FindValue(↓s: string ↓x: char): int
  var i: int
begin
  i := n
  while i > 0 do
    if s[i] = x then
      return i
    end -- if
    i := i - 1
  end -- while
  return 0
end FindValue
```

Das Ergebnis des Suchvorganges wird hier nicht (wie bisher) durch einen Ausgangsparameter an den Rufer übergeben, sondern über den Namen des Algorithmus

Verwendungsbeispiel:

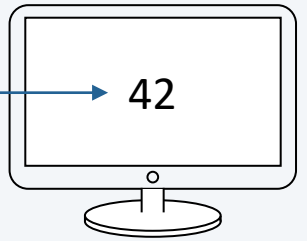
```
if FindValue(↓s ↓'H') > 0 then
   ...
end
```

Geräteschnittstellen

Read hat einen Ausgangsparameter, weil der gelesene Wert einer Variable zugeordnet wird

Dienen zur Kommunikation mit Eingabe- und Ausgabegeräten

```
…
Read(↑x)
y := 6 * x
Write(↓y)
…
```



Anweisung Read liest von einem Eingabegerät den nächsten Wert und stellt diesen als aktuellen Wert der Variablen zur Verfügung

```
Read(↑i)
```

```
Read(↑r)
```

```
var
  i: int
  r: real
```

Anweisung Write schreibt die aktuellen Werte der Ausdrücke auf ein Ausgabegerät

```
Write(↓i)
```

```
Write(↓i ↓42)
```

```
Write(↓42)
```

```
Write(↓i ↓42 ↓'B')
```

1.6 Darstellungsarten von Algorithmen

Es gibt eine Vielzahl von Vorschlägen, eine Auswahl davon sind:

- Grafische Darstellungsformen: Ablaufdiagramm, Struktogramm
- Text-basierte Darstellungsformen: Stilisierte Prosa, Pseudocode, Programmiersprache

Aufgabenstellung für Beispiel:

- Suchen eines Zeichens x in einer Zeichenkette s
- gegeben: Zeichenkette s mit n Zeichen
  Länge der Zeichenkette `n ≥ 1, n = Length(`$\downarrow$ s)
  gesuchter Wert x
- gesucht: Index `i`, so dass
  `i > 0 and s[i] = x` wenn $x \in s$
  `i = 0` wenn $x \notin s$

Stilisierte Prosa

Algorithmus

S1: *Initialisierung*: Setze *i* auf 1

S2: *Gefunden*? Wenn das *i*-te Zeichen von *s* gleich *x* ist, gehe zu S4, sonst erhöhe *i* um 1

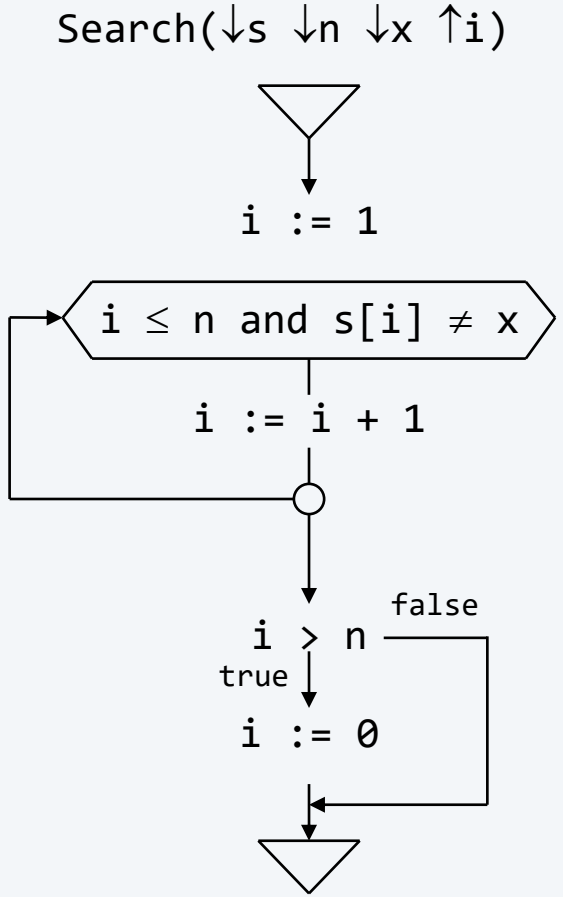
S3: *Ende der Zeichenkette*? Wenn $i \leq n$ ist, gehe zurück zu S2, sonst setze i auf 0

S4: *Fertig*! Ende des Algorithmus

- kommt ohne Formalismen aus
- für den Laien (scheinbar) leicht verständlich
- Ablaufstruktur des Algorithmus ist nur schwer erkennbar

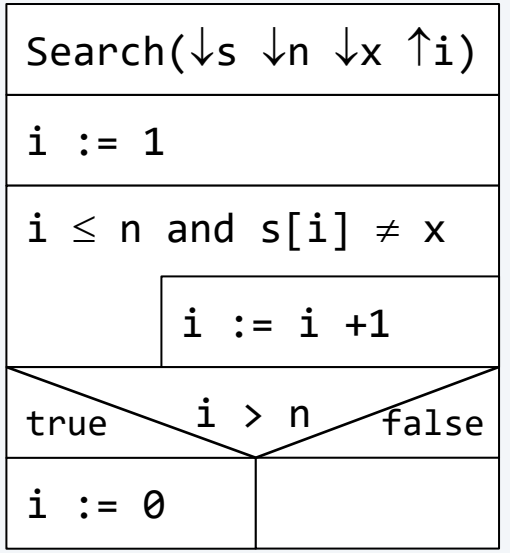
Ablaufdiagramme

- Verwendung halbgrafischer Formalismen
- Eindeutigkeit besser herstellbar
  bei Verwendung genormter Symbole
- Ablaufstruktur gut erkennbar
- Verleitung zu unbeschränkten
  Ablaufstrukturen
- Ohne Werkzeuge schwierig zu
  erstellen und zu pflegen
- Verschiedene Ausprägungen der einzelnen
  Symbole wie Wiederholungen und
  Verzweigungen



Struktogramme

- auch Nassi-Shneiderman-Diagramme genannt
- Verwendung halbgrafischer Formalismen
  (in Analogie zu Ablaufdiagrammen)
- Wenige genormte Symbole
- Ablaufstruktur gut erkennbar
- Unbeschränkte Ablaufstrukturen nicht möglich
- Ohne Werkzeuge kaum zu erstellen und
  zu pflegen (Beschränkung auf kleine Diagramme)
- Veraltete Darstellungsweise



Pseudocode

- an Programmiersprachen angelehnt, aber einfacher (Beseitigung von syntaktischem Ballast)
- Typen für Variablen/Parameter können weggelassen werden

```
Search(↓s: string ↓n: int ↓x: char ↑i: int)
begin
  i := 1
  while (i ≤ n) and (s[i] ≠ x) do
    i := i + 1
  end
  if i > n then
    i := 0
  end
end Search
```

Programmiersprachen

- Programmiersprache hat eine eindeutige Syntax
- wenig Freiheit in der Formulierung, oft schwer zu lesen
- Präziseste und vollständigste Möglichkeit der Algorithmenbeschreibung

Pascal

```
PROCEDURE Search(s: STRING; n: INTEGER; x: CHAR; VAR i: INTEGER);
BEGIN
  i := 1;
  WHILE (i <= n) AND (s[i] <> x) DO
    i := i + 1;
  IF i > n THEN
    i := 0
END; (* Search *)
```

C

```
int search(char s[], int n, char x) {
  int i = 0;
  while (i < n && s[i] != x)
    i++;
  if (i == n)
    i = -1;
  return i;
} // search
```

Welche Darstellungsart wofür?

- Ablaufdiagramm/Struktogramm: für visuelle orientierte Persönlichkeiten
- Pseudocode: halbformal, kurz, präzise, sprachunabhängig, für Algorithmenbücher und Vorlesungsunterlagen
- Programmiersprache: zur Übersetzung und Ausführung auf einem Computer

Kriterien	S. Prosa	Ablaufdiagr.	Struktorgr.	Pseudocode	Pgm.Sprache
Lesbarkeit	-	++	++	+	+/-
Schreibaufwand	--	+/-	--	+	-
Eindeutigkeit	--	+	+	+	++
Strukturierbarkeit	--	-	++	+	+/-
Datendarstellung	--	-	-	+	++
Flexibilität	++	+	+	+	--

1.7 Struktur von Algorithmen

(wieviele Schleifen, Verzweigungen etc.)

Die Struktur eines Algorithmus ist von größter Wichtigkeit für seine Qualität

Fragen zur Gestaltung der Struktur eines Algorithmus sind daher von zentraler Bedeutung

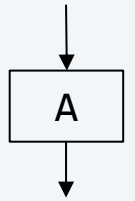
Daher stellen sich folgende Fragen:

- Welche Bausteine man jedenfalls benötigt, um alle denkbaren Algorithmen formulieren zu können?
- Welche Bausteine wirken sich (bei richtiger Verwendung) positiv auf die Qualität eines Algorithmus aus?
- Welche Bausteine wirken sich negativ auf die Qualität eines Algorithmus aus?
- Wie kann man die Strukturqualität von Algorithmen messen?

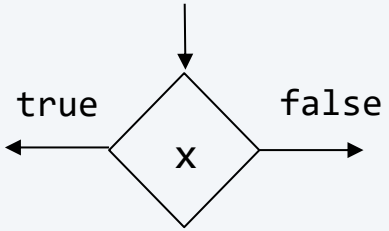
Unbeschränkte Ablaufstrukturen

Alle Konstruktionen zur Ablaufsteuerung lassen sich auf zwei Elemente zurückführen

- elementare Aktion mit
  einem Eingang und einem Ausgang



- binäre Verzweigung mit
  einem Eingang und zwei Ausgängen



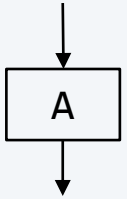
Probleme:

- keine Übereinstimmung zw. statischer u. dynamischer Struktur
- Fremdheit und Vielfalt der Muster
- Quadratisches Wachstum der Pfade

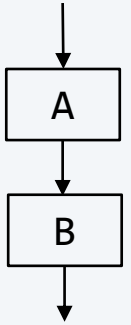
Beschränkte Ablaufstrukturen

Beschränkung der Ablaufstrukturen auf wenige Elemente

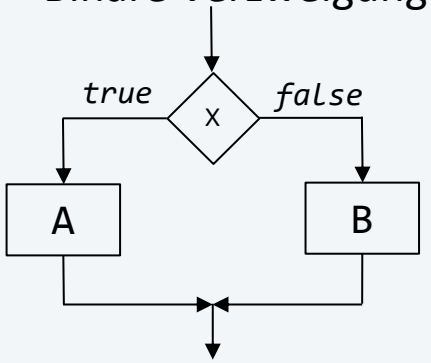
Elementaraktion



Sequenz



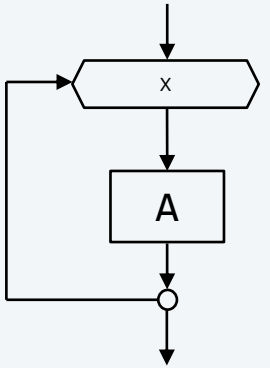
Binäre Verzweigung



wird wieder zusammengeführt

eigentlich egal welche Schleife

Abweisschleife

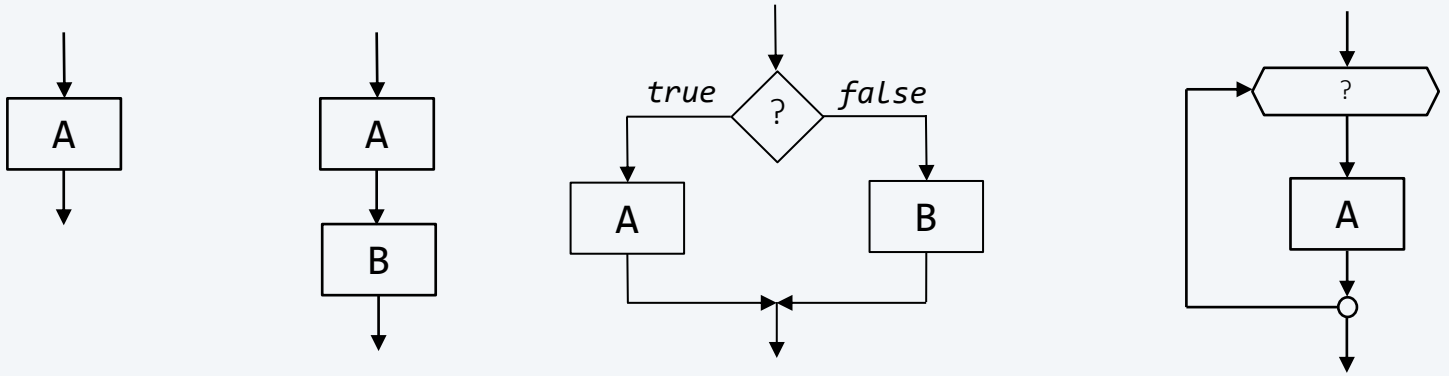


Diese Elemente reichen aus – jedoch sind weitere Elemente sinnvoll

D-Diagramme (benannt nach dem Erfinder E. W. Dijkstra)

D-Diagramme sind Ablaufdiagramme, in denen nur die fünf Muster Elementar-anweisung, Sequenz, Verzweigung (2x) und Abweisschleife vorkommen:

1. Die Elementaranweisung ist ein D-Diagramm.
2. Wenn A und B D-Diagramme sind, dann sind auch



D-Diagramme.

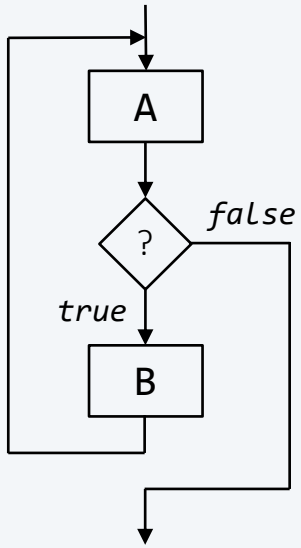
3. Nichts anderes ist ein D-Diagramm.

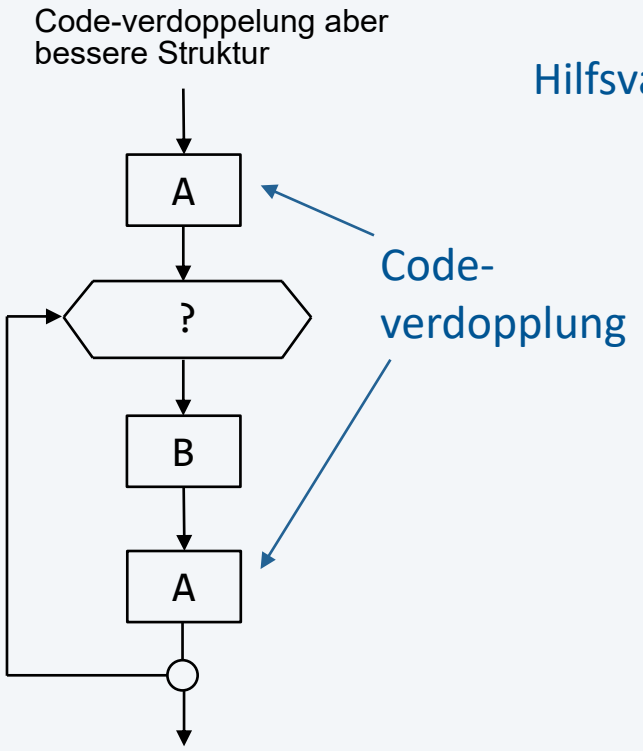
Diese Konstrukte bilden die Basis der strukturierten Programmierung

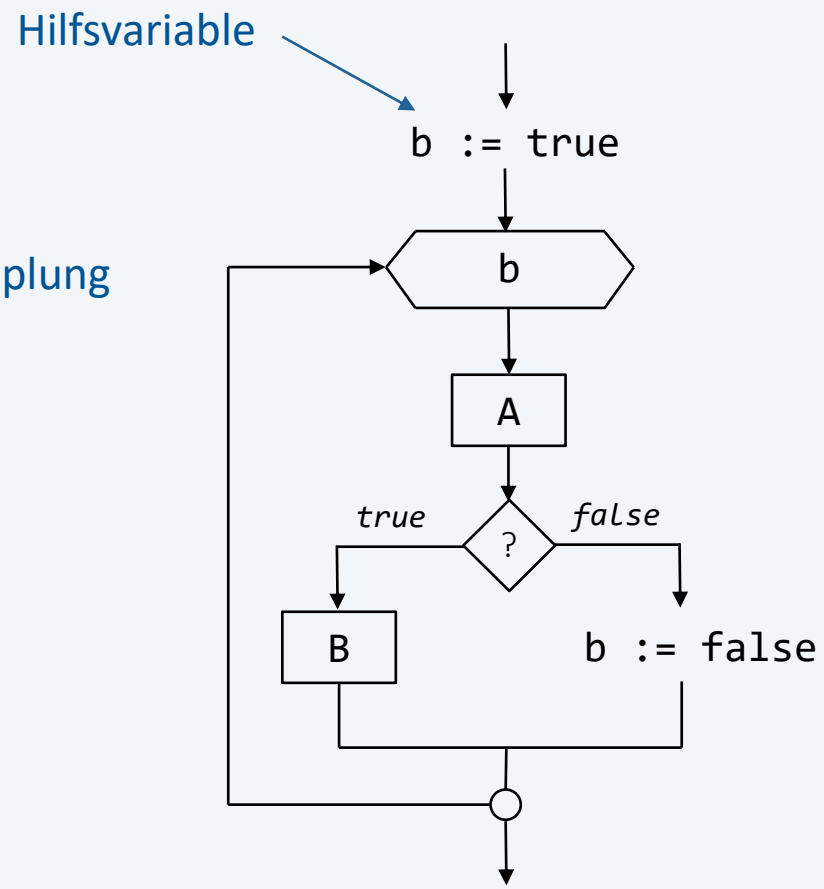
Zu jedem Ablaufdiagramm gibt es ein äquivalentes D-Diagramm

D-Diagramme

Beispiel: Transformation einer unbeschränkten Ablaufstruktur in ein D-Diagramm durch Codeverdopplung oder durch Hilfsvariablen

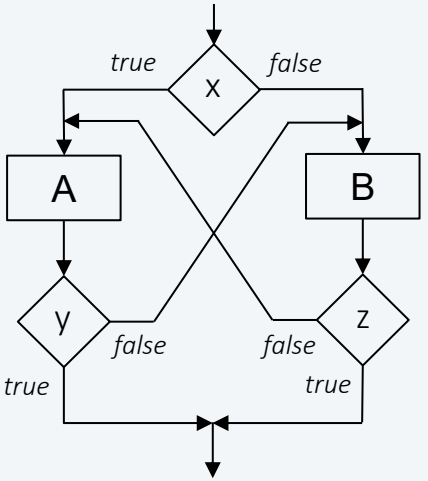






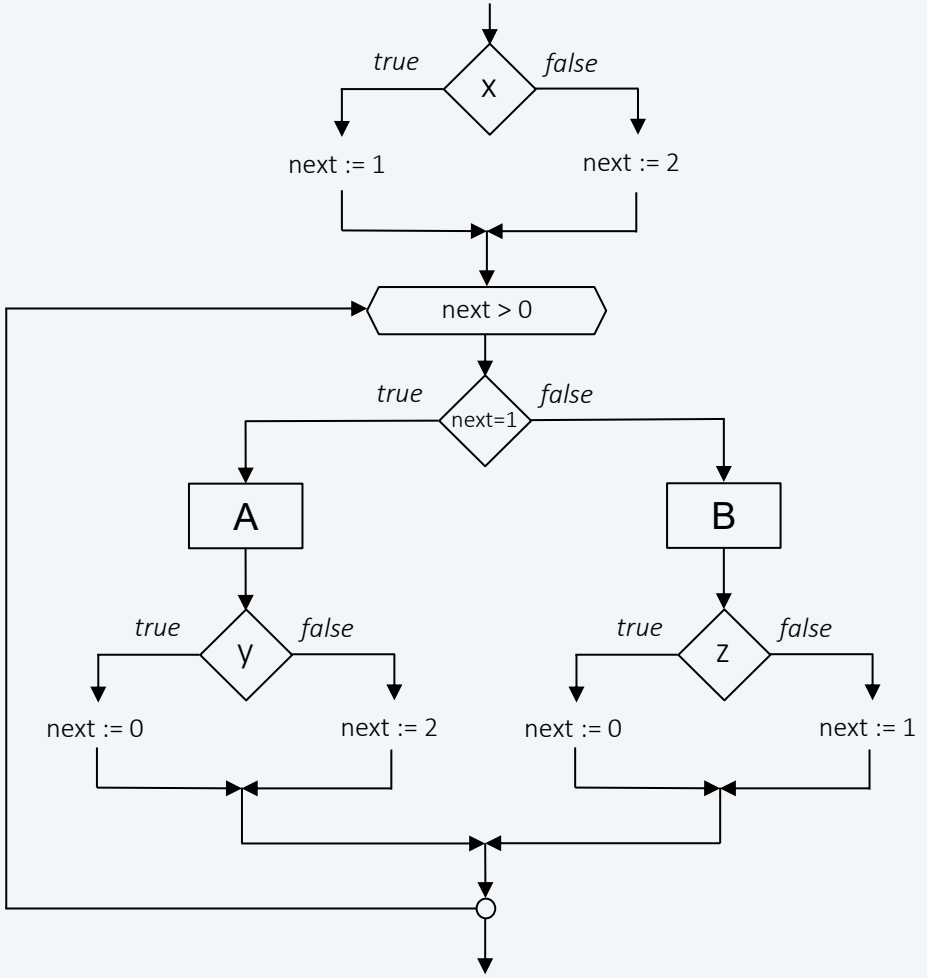
D-Diagramme

Beispiel: Transformation einer Kreuzstruktur



unstrukturiert aber leichter verständlich

nicht notierbar ohne goto Anweisungen



strukturiert aber länger

Strukturierte Programmierung

Vorteile:

- Im Wesentlichen lineares Wachstum der Schwierigkeit mit der Programmlänge
- Gute Korrelation zwischen statischer und dynamischer Struktur
- Einfachheit und Bekanntheit der Muster

Nachteile:

- Unbequemlichkeit
- Manchmal (selten) auch Unklarheit durch Aufblähung
- Ineffizient: durch Codeverdopplung mehr Schreibaufwand, durch überflüssige Verzweigungen mehr Ausführungszeit

D-Diagramm-Elemente sind die Grundbausteine der strukturierten Programmierung

Strukturkomplexität *(V)*

- Da die Qualität eines Algorithmus essentiell von der Ausprägung seiner Struktur abhängig ist, möchte man ein Maß für die Strukturkomplexität eines Algorithmus (um Programmstrukturen hinsichtlich ihrer Qualität miteinander vergleichen zu können)
- Viele Vorschläge, einfaches Maß: Strukturkomplexität *V* nach McCabe
- Annahme von McCabe: Komplexität hängt nur von den Verzweigungen und nicht von der Ausgestaltung der Aktionen ab

Axiome (festgelegt, keine Naturgesetze)

1. V = 1 für Algorithmus mit 1 Pfad
2. Hinzufügen einer binären Verzweigung erhöht V um 1

- Strukturkomplexität *V* = 1 + *Anzahl der binären Verzweigungen*
- Strukturkomplexität sollte ≤ 10 sein

bei hoher Strukturkomplexität braucht man viele Testfälle

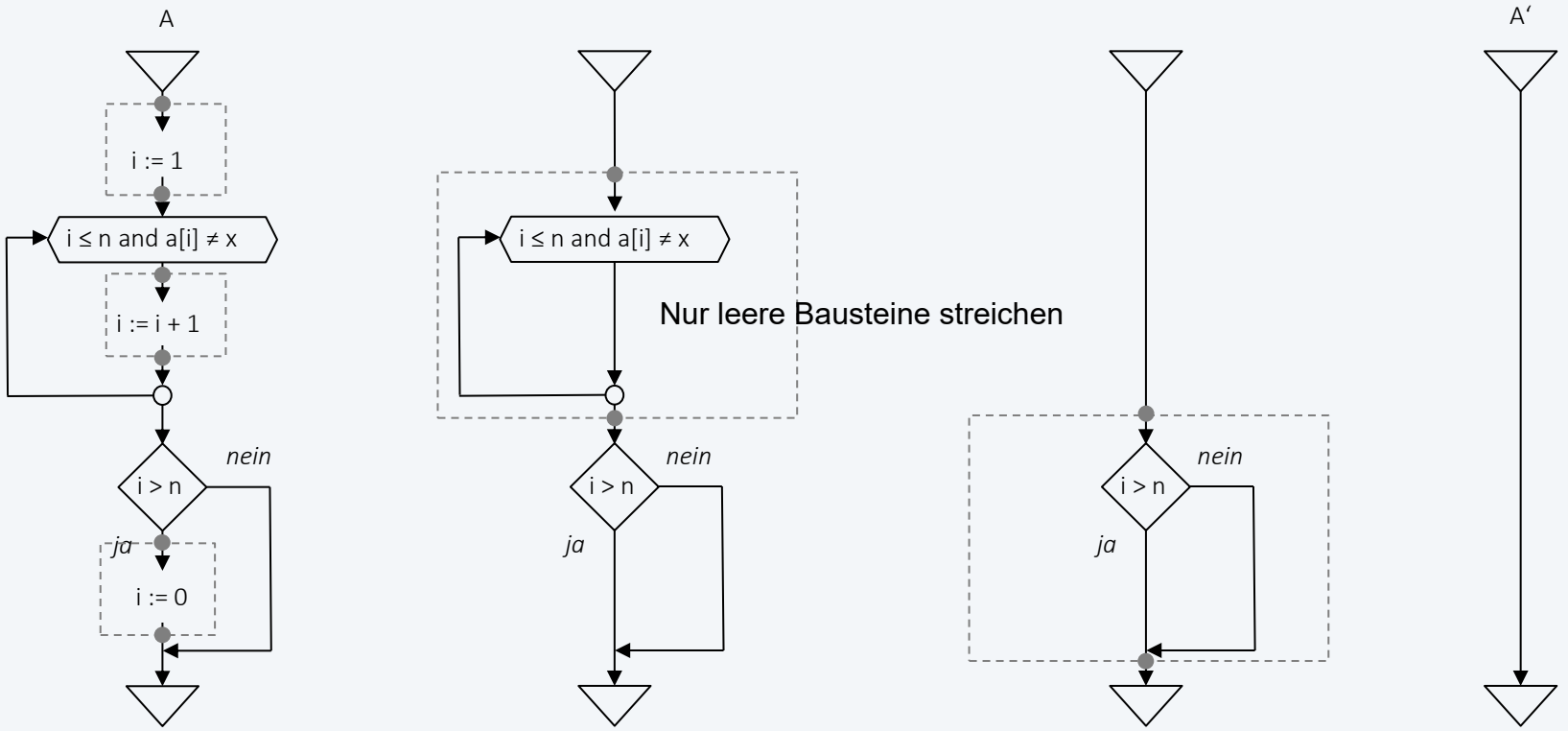
Essentielle Strukturkomplexität (*EV*)

- Strukturkomplexität *V* nach McCabe ist kein Maß, das Auskunft über die Strukturiertheit/Unstrukturiertheit eines Algorithmus gibt
- Um auch dafür ein Maß zu haben, wurde die essentielle Strukturkomplexität *EV* eingeführt

> Definition (Essentielle Strukturkomplexität, *EV*)
>
> *EV*(*A*) eines Algorithmus *A* ist die Strukturkomplexität *V*(*A*'), wobei *A*' der Algorithmus ist, der sich aus *A* durch Reduktion aller D-Diagrammanteile zu einzelnen Knoten ergibt

- Berechnung von EV:
  1. Berechne V nach McCabe
  2. Reduziere alle D-Diagrammanteile bis nichts mehr zu reduzieren ist
  3. Von dieser neuen Struktur berechne wieder V nach McCabe = EV

Strukturkomplexität (Beispiele)

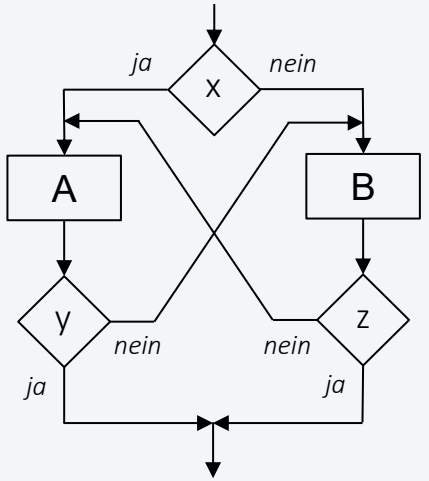


V(A) = 3 (=1 + 1 (Abweisschleife) + 1 Verzweigung) **V(A‘) = 1**

EV(A) = 1

EV ist ein Maß für die strukturierte Programmiertheit. 1 bedeutet, dass nur D-Diagramme verwendet wurden und das Programm somit maximal strukturiert ist

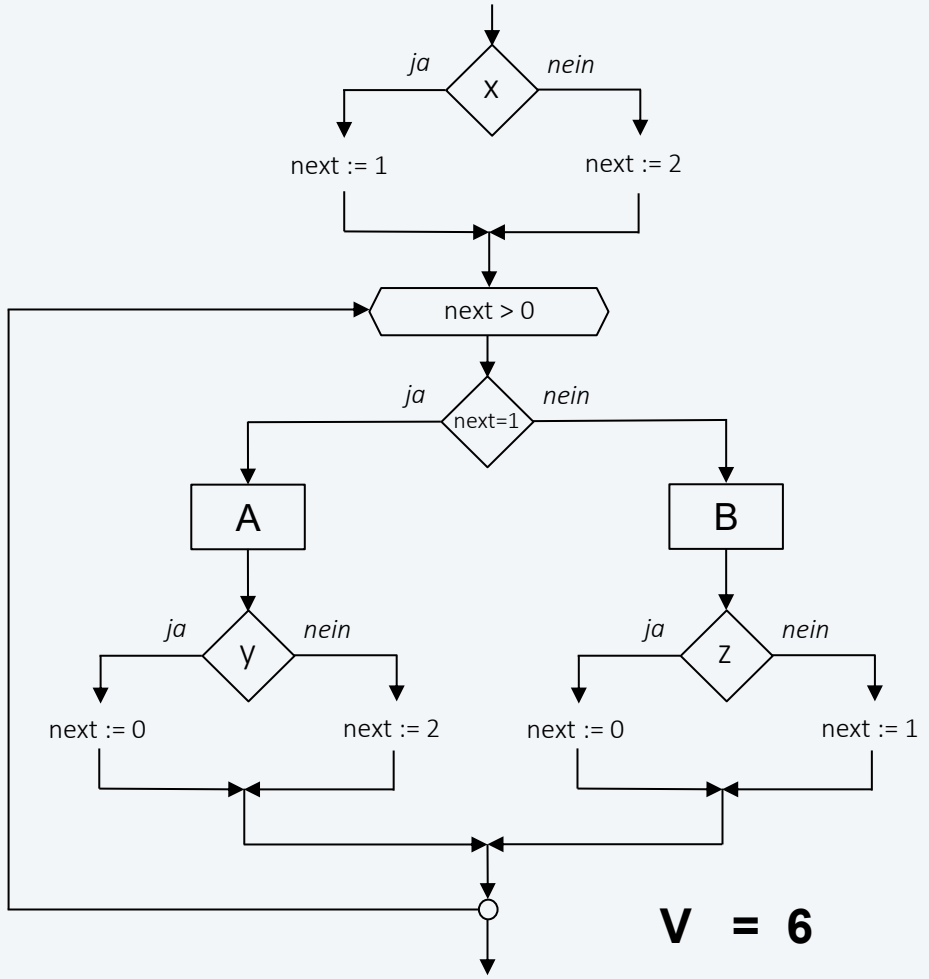
Strukturkomplexität (Beispiele)



V = 4

EV = 4

Kein D-Diagramm verwendet, somit wird nichts weggestrichen



V = 6

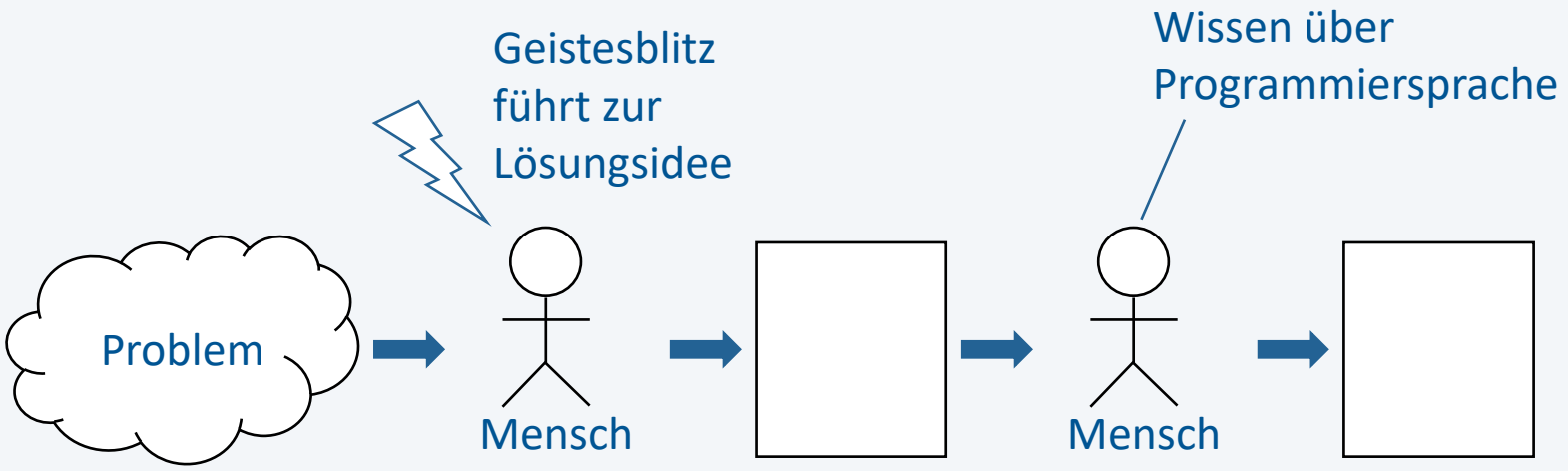
EV = 1

Wie strukturiert man programmiert, ist eine individuelle Entscheidung, aber man muss sich bewusst dafür entscheiden

1.8 Algorithmen und Programme

- Programme sind in einer Programmiersprache formulierte Algorithmen
- Jeder Algorithmus kann in ein Programm transformiert werden
- Alle Forderungen an Algorithmen treffen auch auf Programme zu

Von der Aufgabe über den Algorithmus zum Programm



bekannte Probleme löst Chatgpt weils die schon 100 Mal gibt
Für komplexe Probleme muss man selbst den Geistesblitz haben

Möglichkeiten für die Ausführung eines Programms

Programme, z.B. in Pascal, müssen in eine Form gebracht werden, die vom Computer verstanden wird.

Pascal-Programm → Compiler → Ausführbares Programm

Eingaben ↓

Ausgaben ↓

Maschine kann nicht nachfragen Anweisungen glasklar

Vorteil: einfach
Nachteil: es gibt unterschiedliche Maschinen: mac, windows, linux etc., man braucht einen eigenen compiler

Java-Programm → Compiler → Byte Code → Interpretierer

Vorteil bei Compiler: er erkennt Fehler frühzeitig

jede Plattform hat eigenen Interpretierer

Python-Programm → Interpretierer

Interpretierer übersetzt nicht in Maschinensprache, sondern führt das Programm direkt im eigenen Programm aus

Beschreibung von Programmiersprachen

Die Schreibweise von Programmtexten ist durch Syntaxregeln definiert und muss exakt eingehalten werden

Die Regeln zur Beschreibung der Syntax einer Sprache können auf verschiedene Arten beschrieben werden:

- Syntaxdiagramme

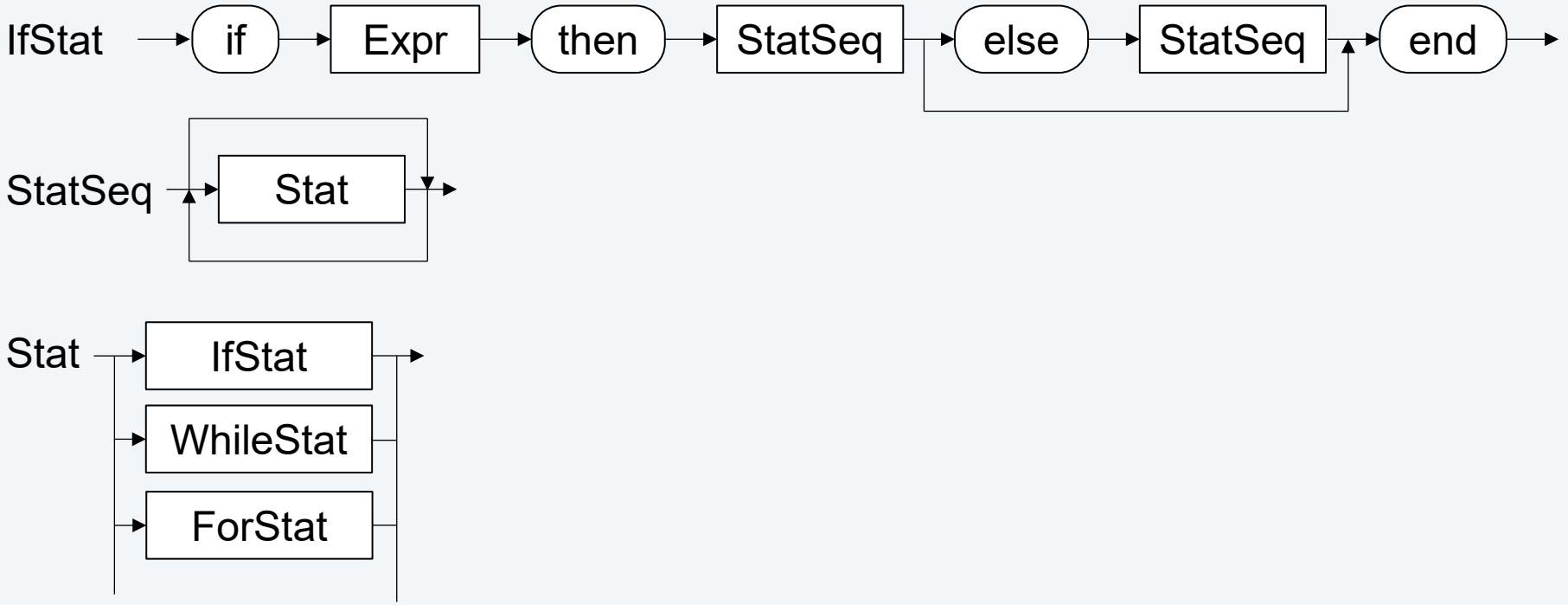
- Grammatiken

```
IfStat  = "if" Expr "then" StatSeq [ "else" StatSeq ] "end".
```

- Automaten

Bestandteile von Syntaxdiagrammen

- Terminalsymbole (T), z.B. if + :=
- Nonterminalsymbole (NT), z.B. IfStat StatSeq Expr
- Ableitungsregeln
  - Für jedes NT-Symbol gibt es eine RegelPfeile führen vom Anfang zum Ende und verbinden die Symbole



Grammatik in Extended Backus-Naur-Form (EBNF)

- Jede Sprachregel für ein NT-Symbol hat die Form
  NT = Regel.
- Regeln bestehen aus
  - Sequenz A B C
  - Alternative A | B | C
  - Option A [B] C
  - Iteration A {B} C
  - Klammern A (B|C) D
- Beispiele für Syntaxdiagramme in EBNF:

```
IfStat     = "if" Expr "then" StatSeq [ "else" StatSeq ] "end".
StatSeq    = { Stat } .
Stat       = IfStat | WhileStat | ForStat | ... .
```