Algorithmen & Datenstrukturen I

WS 2002/03

Prof. Dr. Stefan Fischer

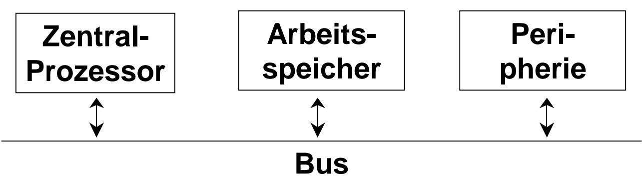
3. Imperative Algorithmen

Inhalt

1. **Einführung**
2. **Variable, Zuweisung und Komplexe Anweisungen**
3. **Arrays**
4. **Prozeduren und Funktionen**

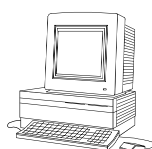
Erinnerung: 1.2 von-Neumann-Rechner

Architekturkonzept von John von Neumann, 1946:



Imperative Programmierung in Kurzform:

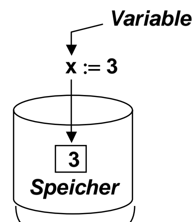
- Anweisungen zum schrittweisen Vorgehen in der Berechnung.
- Zwischenergebnisse werden abgespeichert und bei Bedarf aus dem Speicher wiedergeholt



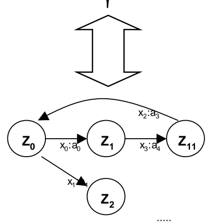
Imperative Konzepte: informeller Überblick (1/2)

- **Variable:**
  Abstraktion eines Speicherplatzes; ein Wert eines gegebenen Datentyps kann gespeichert und beliebig oft gelesen werden, solange nicht ein neuer Wert gespeichert und der alte damit überschrieben wird.

  Achtung! Zweierlei Bedeutung von "Variable":
  1. in Mathematik, Logik, funkt. Programmierung:
     Variable = Platzhalter für (konstanten) Wert
  2. hier: Variable = Speicherplatz (Abstraktion)



- **Zustand:**
  Abstraktion des Speicherinhalts: Gesamtheit der momentanen Werte aller Variablen, ändert sich durch die Ausführung von Anweisungen.

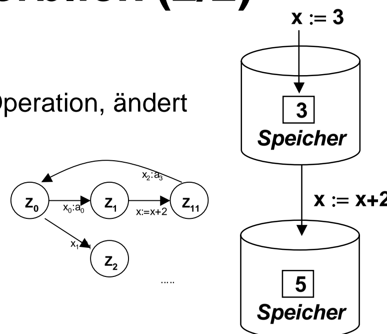


Imperative Konzepte: informeller Überblick (2/2)

- **Anweisung oder Befehl:**
  Vorschrift zur Ausführung einer Operation, ändert i.a. den Zustand (z.B. x. := x+2)

- **Prozedur:**
  Abstraktion einer Anweisung mit definierter Schnittstelle für die Eingabe von Werten und Variablen, ändert i.allg. den Zustand.

  Spezialfall Funktion
  > liefert Rückgabewert



Beispielfunktion:
buchung(kVar: Konto, bVar: Betrag): Betrag
Seiteneffekt: Änderung des Kontostandes
Ergebnis: nachfolgender Kontostand

Grundlegende Datentypen (1/3)

Definition Datentyp:

Unter einem Datentyp versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit.

Abstrakter Datentyp:

Schwerpunkt liegt auf den Eigenschaften, die die Operationen und Wertebereiche besitzen

> Kapitel 4

Konkreter Datentyp

Darstellung in einer Programmiersprache steht im Vordergrund

> Hier (eigentlich) benötigt!

Grundlegende Datentypen (2/3)

Häufig: Überführung mathematischer Zahlenmengen etc. in *Datentypen* (inklusive Operationen)

Grundlegende Datentypen (1/2):

bool: die Boole'schen Werte IB = **{O, L}** mit ihren Operationen $\vee$, $\wedge$, $\neg$, ...

int: ganze Zahlen {*minint*,...,-2,-1,0,1,2,...,*maxint*} $\subseteq \mathbb{Z}$

Meist im binären Zahlensystem dargestellt. *minint* und *maxint* sind im Absolutbetrag i.a. etwa gleich groß . . . und zwar ziemlich groß: bei 32 Bit-Darstellung 2^{31} -1 = 2147483647.

Die Rechengesetze (Assoziativgesetze der Addition und Multiplikation, Distributivgesetze u.s.w.) gelten nicht, wenn man über die Darstellungsgrenze gerät.

Grundlegende Datentypen (3/3)

Grundlegende Datentypen (2/2)

real, auch **float**: reelle Zahlen IR, durch *Gleitpunktzahlen* $z=m*b^e$ dargestellt
$m \in$ IR *Mantisse*, $0 \leq m < 1$, $b \geq 2 \in$ IN *Basis*, $e \in \mathbb{Z}$ *Exponent*.

Die Darstellung erschließt einen sehr großen darstellbaren Zahlenbereich mit konstanter relativer Darstellungsgenauigkeit.

Die Rechengesetze gelten wegen der Rundungsfehler nicht streng. Es macht keinen Sinn, Gleitpunktzahlen auf Gleichheit zu prüfen: statt $x = y$ nehme man $|x-y| < \varepsilon$

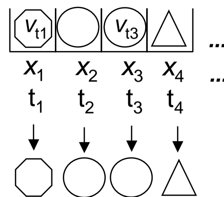
char: ein Alphabet darstellbarer Zeichen 0,...,9,A,..., *Z*,*a*,...,*z*,(,),+,...

Mit Vergleichsoperationen $\leq,<,\geq,>,=,\neq$ und Operationen *ord*:char $\rightarrow$ int und *chr*: int $\rightarrow$ char, die die Zuordnung zur Nummer des Zeichens im Alphabet herstellen.

4.2 Variable, Zuweisung und Komplexe Anweisungen

Variable x : t

- Sei X = { x_1 : t_1, ... , x_n : t_n } eine endliche Menge von Variablen
- t = $\tau(x)$ Typ von x
- $v = w(x)$ Wert von x
- $v \in W(t)$ Wertemenge von t



Deklaration var $x_1 : t_1, \ldots, x_n : t_n$

- Vereinbarung der angegebenen Variablen
- Dient der *Bereitstellung des Speicherplatzes*
- Notation:
  var x,y, ... : t für **var** x : t , y : t, ... bei gleichem Typ

Zustand

Zustand = Belegung aller Variablen (mit Werten der entsprechenden Typen) zu einem Zeitpunkt

Modellierung: $\sigma = \{ x_1 \rightarrowtail v_1 , \ldots , x_n \rightarrowtail v_n \}$

$x_i \in X$, $v_i \in W(t_i)$ für $i \in \{1, \ldots ,n\}$.

Tabellarisch:

x_1	...	x_n
v_1	...	v_n

Mathematisch: Abbildung $\sigma : X \rightarrow$ =: W

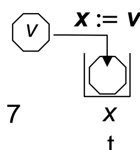
wobei $\sigma(x_i) \in W(t_i)$ für alle $i \in \{1, \ldots ,n\}$

D.h. $\sigma(x)$ ist die aktuelle Belegung der Variablen x

Zuweisung

Zuweisung

- **Syntax:** $\boldsymbol{x := v}$ *mit* $v \in W(\tau(x))$
- **Grundanweisung**: „nachher“ gilt $w(x) = v$
- **Notation**: wir schreiben x statt $w(x)$: $x = 7$ statt $w(x) = 7$



Semantik (formal):

Ist σ: $X \rightarrow W$ ein Zustand und wählt man eine Variable $x \in X$, sowie einen Wert v vom passenden Typ, so ist der *transformierte Zustand* $\sigma_{\langle x \leftarrow v \rangle}$ wie folgt definiert:

$$\sigma_{\langle x \leftarrow v \rangle}(y) = \begin{cases} v, & \text{falls } x = y \\ \sigma(y) & \text{sonst} \end{cases}$$

Semantikfestlegung einer Anweisung $\underline{a} \in A$ allgemein :
$[\![\;]\!]: (X \rightarrow W) \rightarrow A \rightarrow (X \rightarrow W)$ mit $[\![\underline{a}]\!](\sigma) = \sigma'$

Semantikfestlegung der Zuweisung: $[\![\boldsymbol{x := v}]\!](\sigma) = \sigma_{\langle x \leftarrow v \rangle}$

Auswertung imperativer Grundanweisungen (1/5)

Vorbemerkung Terme:
Terme sind (mathematisch) bedeutsame Zeichenfolgen und haben einen Typ (> Auswertung)

Beispiele: *4*(3-2)+3* oder *13-Sqrt(2)+3*

Beispielhafte Definition von int-Termen:

1. Die int-Werte ...,-2,-1,0,1,2,... sind int-Terme
2. Sind *t,u* int-Terme*, so auch* (*t+u*)*,* (*t-u*)*,* (*t*∗*u*)*,* (*t* ÷ u)
3. Ist b ein bool-Term, und sind t, u int-Terme, so sind auch
   - *if b then t else u fi* und
   - *if b then t fi* int-Terme
4. Nur die mit diesen Regeln gebildeten Zeichenketten sind int-Terme

> Weitere Definitionen (bool-Terme, real-Terme etc.) analog

Auswertung imperativer Grundanweisungen (2/5)

Definition Terme mit Unbestimmten (Variablen)

Zunächst Einführung von zwei abzählbar unendliche Mengen von Symbolen (sog. Unbestimmte > Variablen)

- *x,y,z,...* vom Typ int
- *a,b,c,...* vom Typ real
- *p,q,r,...* vom Typ bool

Zur eigentlichen Definition:

1. *Erweiterung der Definition der int-Terme (s.o.)*
   1b. Die int-Unbestimmten *x,y,z,...* sind int-Terme
   Damit: x-2, 2∗x-1, (x+1) ∗(y-1) etc. sind int-Terme
2. *Erweiterung der Definition der bool-Terme (s.o.)*
   ?. Die bool-Unbestimmten p,q,r,... sind bool-Terme
   Damit: p, $p \vee q$, $p \wedge r$, $\neg r$ etc. sind bool-Terme
3. *etc.*

Auswertung imperativer Grundanweisungen (3/5)

Definition Ausdruck:
Ausdrücke entsprechen im Wesentlichen den Termen Sprache, jedoch stehen an Stelle der *Unbekannten Variablen*.

=> Die Auswertung von Termen mit Variablen ist *zustandsabhängig*. An die Stelle der Variablen wird ihr aktueller Wert gesetzt.

Beispiel: Für einen gegebenen Term "2∗x+1" ist der Wert im Zustand σ durch 2∗σ(x)+1 festgelegt.

Auswertung imperativer Grundanweisungen (4/5)

Festlegung: Der derart bestimmte Wert eines Ausdrucks $t(x_1,x_2,...,x_n)$ wird mit $\sigma(t(x_1,x_2,...,x_n))$ bezeichnet.

Beispiel: σ(2∗x+1) = 2∗σ(x)+1

=> Diese Festlegung erlaubt Wertzuweisungen mit Variablen auf der rechten Seite

$$y := t(x_1, x_2, ..., x_n)$$

Der *transformierte Zustand* hierfür ist wie folgt festgelegt:

$$[\![y := t(x_1, x_2, ..., x_n)]\!] (\sigma) = \sigma_{\langle y \leftarrow \sigma(t(x_1, x_2, ..., x_n)) \rangle}$$

Auswertung imperativer Grundanweisungen (5/5)

Beispiele: Transformationen zweier elementarer Anweisungen α_1 und α_2:

- $\alpha_1 = (x := 2*y+1)$
  => Transformation in $[\![\alpha_1]\!](\sigma) = \sigma_{\langle x \leftarrow 2 * \sigma(y)+1 \rangle}$
- $\alpha_2 = (x := 2*x+1)$
  => Transformation in $[\![\alpha_2]\!](\sigma) = \sigma_{\langle x \leftarrow 2 * \sigma(x)+1 \rangle}$

Achtung: α_2 definiert keine rekursive Gleichung für x !

Anmerkung:
Wertzuweisungen sind die einzigen *elementaren Anweisungen* imperativer Algorithmen. Aus ihnen werden komplexe Anweisungen zusammengesetzt, aus denen imperative Algorithmen entstehen.

Komplexe Anweisungen (1/7)

Komplexe Anweisungen iterative Algorithmen bilden eine Untermenge der in Kapitel 2 behandelten intuitiven *Algorithmenbausteine* und *Konstruktionsprinzipien (Zuweisungen sind als elementare Operationen zu betrachten):*

1. *Sequentielle Ausführung*
2. *Bedingte Ausführung*
3. *Schleife*
4. *(Parallele Ausführung)*
5. *(Unterprogramm)*
6. *(Rekursion)*

} Wegen ihrer Bedeutung für die Abfolge der Ausführung elementarer Operationen bzw. die *Ablaufsteuerung* auch als *Kontrollstrukturen* bezeichnet.

Komplexe Anweisungen (2/7)

1. ***Sequenz***

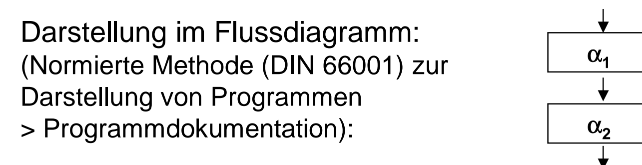
Definition: Sind α_1 und α_2 Anweisungen, so auch $\alpha_1 ; \alpha_2$

Anschaulich: "Führe erst α_1 , dann α_2 aus"

Semantikfestlegung: $[\![\alpha_1 ; \alpha_2]\!](\sigma) = [\![\alpha_2]\!]([\![\alpha_1]\!](\sigma))$

=> Hintereinanderausführung der beiden Funktionen, welche die einzelnen Schritte definieren

Darstellung im Flussdiagramm:
(Normierte Methode (DIN 66001) zur Darstellung von Programmen
> Programmdokumentation):



3. Imperative Alg. 3.2 Var., Zuw. & AS

Komplexe Anweisungen (3/7)

2. ***Bedingte Ausführung (Selektion)***

Definition: Sind α_1 und α_2 Anweisungen und ***P*** ein Boole'scher Ausdruck, so ist auch

if *P* then α_1 else α_2 fi

eine Anweisung.

Anschaulich: "Falls *P* gilt, führe α_1 , ansonsten α_2 aus"

Semantikfestlegung:

$$[\![\text{if } P \text{ then } \alpha_1 \text{ else } \alpha_2 \text{ fi}]\!](\sigma) = \begin{cases} [\![\alpha_1]\!](\sigma), \text{ falls } \sigma(P) = \text{true} \\ [\![\alpha_2]\!](\sigma), \text{ falls } \sigma(P) = \text{false} \end{cases}$$

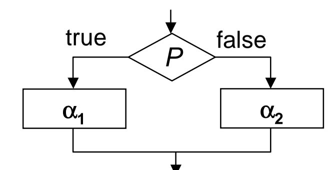
3. Imperative Alg. 3.2 Var., Zuw. & AS

Komplexe Anweisungen (4/7)

2. ***Fortsetzung Bedingte Ausführung (Selektion)***

Voraussetzung: $\sigma(P)$ kann ausgewertet werden (ansonsten ist die Anweisung undefiniert)

Darstellung im Flussdiagramm:



Anmerkungen zum Flussdiagramm:

1. Flussdiagramm beginnt mit [Start] und endet mit [Stop]
2. Ein- und Ausgaben werden beschreiben durch [Eingabe *n*] und durch [Ausgabe *p*], wobei *n* und *p* Variable oder Terme.

Komplexe Anweisungen (5/7)

3. Schleife (Iteration, Wiederholung)

Definition: Ist α eine Anweisung und ***P*** ein Boole'scher Ausdruck, so ist auch

while *P* do α od

eine Anweisung.

Anschaulich: "Solange *P* gilt, führe α aus"

Semantikfestlegung:

$$[\![\text{while } P \text{ do } \alpha \text{ od}]\!](\sigma) = \begin{cases} \sigma, \text{ falls } \sigma(P) = \text{false} \\ [\![\text{while } P \text{ do } \alpha \text{ od}]\!]([\![\alpha]\!](\sigma)), \text{ sonst} \end{cases}$$

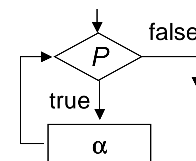
Komplexe Anweisungen (6/7)

3. Fortsetzung Schleife (Iteration, Wiederholung)

Voraussetzung: $\sigma(P)$ kann ausgewertet werden (ansonsten ist die Anweisung undefiniert)

Anmerkung: Die Definition ist rekursiv!

Darstellung im Flussdiagramm:



Komplexe Anweisungen (7/7)

Anmerkungen:

- In existierenden imperativen Programmiersprachen gibt es *fast immer diese* Anweisungen, oft jedoch mehr.
- Programmiersprachen mit nur diesen Sprachelementen sind *bereits universell*. D.h. alle Algorithmen sind formulierbar.
- Die *Syntax der Kontrollstrukturen variiert* natürlich, z.B.
  1. andere Schlüsselworte (z.B. end if *anstelle* fi)
  2. werden i.d.R. Klammerungen verwendet (**while** *P* **do** {...}; ...)
- **while**-Schleifen müssen nicht terminieren (rekursive Definition > häufige Fehlerquelle)

Imperative Algorithmen (1/3)

Vorbemerkung: Verwendung der Datentypen int, bool, real

Imperative Algorithmen haben folgenden Aufbau:

<Programmname>

var *x*,*y*,...: int, *p*,*q*,...: bool; ← **Variablendeklaration**

input: $x_1, x_2, \ldots, x_n$; ← **Eingabevariablen**

α; ← **Anweisung(en)**

output: $y_1, y_2, \ldots, y_m$; ← **Ausgabevariablen**

Semantikfestlegung: Bestimmung einer „Zustandsüberführungsfunktion"!

Imperative Algorithmen (2/3)

Die Semantik eines imperativen Algorithmus ist eine partielle Funktion:

$[\![\text{PROG}]\!] : W_1 \times ... \times W_n \rightarrow V_1 \times ... \times V_m$

$[\![\text{PROG}]\!] (w_1,... ,w_n) \quad = (\sigma (y_1),..., \sigma (y_n))$

wobei $\sigma \quad = [\![\alpha]\!] (\sigma_0)$

$\sigma_0(x_i) \quad = w_i \in W_i$ für $i = 1,...,n$

und $\sigma_0(y) \quad = \bot$ für alle Variablen $y \neq x_i$, $i = 1,...,n$

Wobei folgende Abkürzungen/Konventionen gelten:

PROG	Programmname
$W_1, ..., W_n$	Wertebereiche der Typen von $x_1,...,x_n$
$V_1, ..., V_m$	Wertebereiche der Typen von $y_1,...,y_n$

Imperative Algorithmen (3/3)

Anmerkungen:

- Der Algorithmentext definiert *eine Transformation* auf dem gesamten, *durch die Eingaben initialisierten Zustand*. Seine Bedeutung ist aus den *Werten der Ausgabevariablen* „auszulesen“
- Achtung: falls die Auswertung von α nicht terminiert, ist die Funktion $[\![\text{PROG}]\!]$ *nicht definiert*.

Charakterisierung imperativer Algorithmen:

- Die Ausführung imperativer Algorithmen besteht aus einer Folge von Grundanweisungen (genauer Zuweisungen). Diese Folge wird mittels *Bedingter Ausführung* und *Iteration* durch *Auswertung bool‘scher Ausdrücke* über dem Zustand *konstruiert*.
- Jede Zuweisung definiert eine *elementare Transformation* des Zustands. Die *Semantik* des Algorithmus ist durch die *Kombination all dieser Zustandstransformationen* festgelegt.

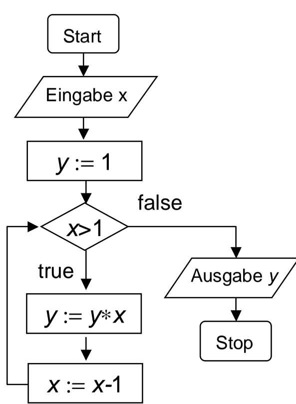
Beispiele Imperativer Algorithmen (1/10)

Fakultätsfunktion n! = 1∗2∗3∗... ∗(n-1) ∗n

Version 1:

FAK1: **var** *x*, *y* : **int**;
input *x*;
y := 1;
while *x* > 1 **do**
y := *y* ∗ *x* ;
x := *x* - 1 **od;**
output *y*

y ist Speicher für Zwischen- und Endergebnis



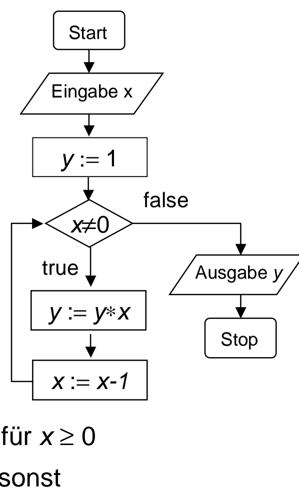
Es gilt: $[\![\text{FAK1}]\!] (\sigma) = \begin{cases} x! \text{ für } x \geq 0 \\ 1 \text{ sonst} \end{cases}$

Beispiele Imperativer Alg. (2/10)

Fakultätsfunktion n! = 1∗2∗3∗... ∗(n-1) ∗n

Version 2 entstehe aus Version 1 durch Veränderung der Abbruchbedingung:

FAK2: **var** *x*, *y* : **int**;
input *x*;
y := 1;
while *x* ≠ **0 do**
y := *y* ∗ *x* ;
x := *x* - 1 **od;**
output *y*



FAK2: "x≠0" anstelle "x>0" $\Rightarrow [\![\text{FAK2}]\!] (\sigma) = \begin{cases} x! \text{ für } x \geq 0 \\ \bot \text{ sonst} \end{cases}$

Beispiele Imperativer Alg. (3/10)

Auswertung der Fakultätsfunktion FAK1

```
FAK1:  var x, y : int;
       input x;
       y := 1;
       while x > 1 do
            y := y * x ;
            x := x - 1 od
       output y
```

- Signatur der Semantikfunktion: $[\![\text{FAK1}]\!]$: int → int
- Resultat der Funktion ist die Belegung der Variablen y im Endzustand: $[\![\text{FAK1}]\!]\,(x{=}v) = \sigma(y)$
- Endzustand σ ist lt. Definition definiert durch $\sigma = [\![\,\alpha\,]\!]\,(\sigma_0)$, wobei α die Folge aller Anweisungen des Algorithmus ist.
- σ_0 ist definiert als $\sigma_0 = (\,x{=}v,\ y{=}\perp)$ in Kurzschreibweise $(v, \perp)$
- Gesucht: FAK1(3)

$$\begin{aligned}\sigma &= [\![\,\alpha\,]\!]\,(\sigma_0) = [\![\,\alpha\,]\!]\,(3,\perp)\\ &= [\![\,y := 1;\ \text{while } x > 1 \text{ do } y := y * x\,;\ x := x - 1 \text{ od}\,]\!]\,(3,\perp)\\ &= [\![\,\text{while } x > 1 \text{ do } y := y * x\,;\ x := x - 1 \text{ od}\,]\!]\,(\,[\![\,y := 1\,]\!]\,(3,\perp)\,)\\ &=: [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,([\![\,y := 1\,]\!]\,(3,\perp))\end{aligned}$$

Beispiele Imperativer Alg. (4/10)

```
FAK1:  var x, y : int;
       input x;
       y := 1;
       while x > 1 do
            y := y * x ;
            x := x - 1 od
       output y
```

$$\begin{aligned}\ldots &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,([\![\,y := 1\,]\!]\,(3,\perp))\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\ (3,\perp)_{\langle y\leftarrow 1\rangle}\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\ (3,\mathbf{1})\\ &= \begin{cases}\sigma \text{ für } \sigma(B) = \text{false}\\ [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,([\![\,\beta\,]\!]\,(\sigma)), \text{ sonst}\end{cases}\\ &= \begin{cases}(3,1), \text{ falls } \sigma(x > 1) = (3{>}1) = \text{ false}\\ [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![y := y * x\,;\ x := x - 1]\!]\,(\sigma)), \text{ sonst}\end{cases}\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![y := y * x\,;\ x := x - 1]\!]\,(3,1))\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![x := x - 1]\!]\ ([\![y := y * x\,]\!]\,(3,1))\,)\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![x := x - 1]\!]\,(3,3)\,)\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(2,3)\end{aligned}$$

Beispiele Imperativer Alg. (5/10)

$$\begin{aligned}\ldots &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(2,3)\\ &= \begin{cases}(2,3), \text{ falls } \sigma(x > 1) = (2{>}1) = \text{ false}\\ [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![y := y * x\,;\ x := x - 1]\!]\,(\sigma)), \text{ sonst}\end{cases}\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![y := y * x\,;\ x := x - 1]\!]\,(2,3)\,)\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,([\![x := x - 1]\!]\,(\,[\![y := y * x\,]\!]\,(2,3)\,)\,)\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![x := x - 1]\!]\,(2,6)\,)\\ &= [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(1,6)\\ &= \begin{cases}(1,6), \text{ falls } \sigma(x > 1) = (1{>}1) = \text{ false}\\ [\![\,\text{while } B \text{ do } \beta \text{ od}\,]\!]\,(\,[\![\beta]\!]\,(\sigma)), \text{ sonst}\end{cases}\\ &= (1,6)\end{aligned}$$

$\Rightarrow [\![\,\mathbf{FAK1}\,]\!]\,(\mathbf{3}) = \mathbf{6}$

```
FAK1:  var x, y : int;
       input x;
       y := 1;
       while x > 1 do
            y := y * x ;
            x := x - 1 od
       output y
```


Beispiele Imperativer Alg. (6/10)

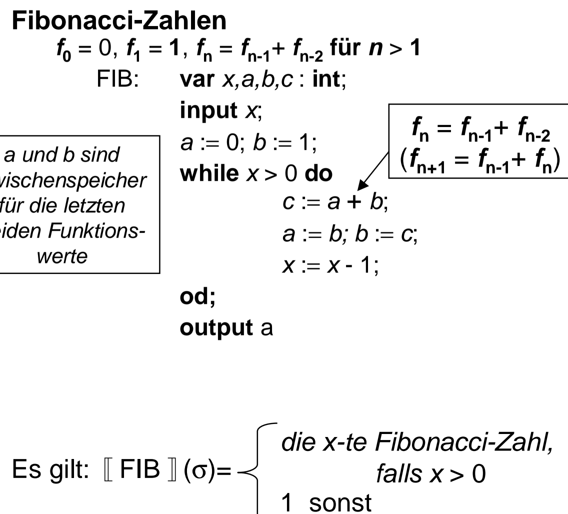
Fibonacci-Zahlen

$\boldsymbol{f_0 = 0,\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2}}$ **für** $\boldsymbol{n > 1}$

```
FIB:  var x,a,b,c : int;
      input x;
      a := 0; b := 1;
      while x > 0 do
              c := a + b;
              a := b; b := c;
              x := x - 1;
      od;
      output a
```

a und b sind Zwischenspeicher für die letzten beiden Funktions-werte

$\boldsymbol{f_n = f_{n-1} + f_{n-2}}$
$\boldsymbol{(f_{n+1} = f_{n-1} + f_n)}$



Es gilt: $[\![\,\text{FIB}\,]\!]\,(\sigma) = \begin{cases}\textit{die x-te Fibonacci-Zahl, falls } x > 0\\ 1 \ \text{ sonst}\end{cases}$

Beispiele Imperativer Alg. (7/10)

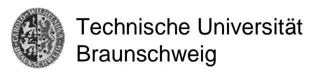
Größter gemeinsamer Teiler *ggT*(*a*,*b*) für *a*,*b* ≥ 0, Version 1

GGT1: **var** ***x*,*y*** **: int;**
input *x*,*y*;
while $x \neq y$ **do** (1)
  while $x > y$ **do** $x := x - y$; **od;** (1.1)
  while $x < y$ **do** $y := y - x$; **od** **od** (1.2)
output *x*

Auswertung ⟦ GGT1 ⟧ (17,5):

# ':='	x	y	while-Schleife
0	17	5	Eintritt 1
1	12	5	Eintritt 1.1
2	7	5	1.1
3	2	5	1.1
4	2	3	Austritt 1.1, Eintritt 1.2
5	2	1	1.2
6	1	1	Austritt 1.2 , Eintritt 1.1 und Austritt 1.1 und 1

Berechnung allein durch Subtraktion ist nicht sehr effizient (Bsp. ggt(2,1000)) Verbesserung durch Division?



Beispiele Imperativer Alg. (8/10)

Größter gemeinsamer Teiler *ggT*(*a*,*b*) für *a*,*b* ≥ 0, Version 2

GGT2: **var** ***x*,*y*,*r*** **: int;**
input *x*,*y*;
$r := 1$;
while $r \neq 0$ **do**
  $r := x \bmod y$; $x := y$; $y := r$ **od**
output *x*

Auswertung ⟦ GGT2 ⟧ (17,5):

# ':='	x	y	r
1	17	5	1
4	5	2	2
7	2	1	1
10	1	0	0

Auswertung ⟦ GGT2 ⟧ (2,1000):

# ':='	x	y	r
1	2	1000	1
4	1000	2	2
7	2	0	0

Beispiele Imperativer Alg. (9/10)

Größter gemeinsamer Teiler *ggT*(*a*,*b*) für *a*,*b* ≥ 0, Version 2 (Fortsetzung)

Was berechnet GGT2 im Falle negativer *x* und *y*?

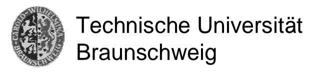
Semantikfunktion von GGT2:

$$
[\![\text{GGT2}]\!](\sigma) = \begin{cases} ggT(x,y) & \text{falls } x,y > 0 \\ \bot & \text{falls } y=0 \\ ggT(|x|,|y|) & \text{falls } x<0 \text{ und } y>0 \\ -ggT(|x|,|y|) & \text{falls } y < 0 \\ y & \text{falls } x=y\neq 0 \text{ oder } x=0,\ y\neq 0 \end{cases}
$$

GGT2: ... $r := 1$;
while $r \neq 0$ do
 $r := x \bmod y$;
 $x := y$;
 $y := r$
od ...

Anmerkung:

Intuitiv ist GGT2 effizienter als GGT1. Aber wie beweist man eine derartige Eigenschaft? Dazu später mehr...



Beispiele Imperativer Alg. (10/10)

Was berechnet der folgende Algorithmus?

XYZ: **var** ***w*,*x*,*y*,*z*** **: int;**
input *x*;
$z := 0$; $w := 1$; $y := 1$;
while $w \leq x$ **do**
  $z := z + 1$; $w := w + y + 2$; $y := y + 2$ **od**
output *z*

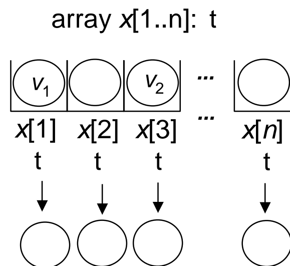
Beispielhafte Auswertung ⟦ XYZ ⟧ :

w	x	y	z
1	0	1	0
1	1	1	0
4		3	1
1	2	1	0
4		3	1

w	x	y	z
1	3	1	0
4		3	1
1	4	1	0
4		3	1
9		5	2

4.3 Arrays

...sind "Variablenfelder" fester Länge



- **Pragmatik:** Verwaltung einer indizierten Menge von *n* Variablen *gleichen Typs*
  => generischer Datentyp
- **Deklaration:**
  var array *x*[*I*]: t mit
  1. *I* endliche Indexmenge (z.B. [1..5]); i.d.R. ist $I \subseteq$ IN
  2. t Typ der Arrayvariablen

  => Array x ≅ Menge von Variablen {x[i]| i ∈ *I* }
- Zugriff auf ein einzelnes Feld: x[*i*], mit i ∈ *I* (bzw. i = Ausdruck, der zu einem Element der Indexmenge ausgewertet werden kann; Beispiel: *y* := *x*[*j*+2])
- Längenänderung zur Laufzeit (i.d.R.) nicht möglich

Verbreitete Spezialfälle

1. ***I* = [1..*n*]:**
   var array *x*[1..*n*]: t ≅ Vektor von Variablen (*x*[1], ... ,*x*[*n*])

   Beispiel:
   var *v*[1..3]: **real** ≅ Vektor über IR³
   v[1] := 1; v[2] := 3; v[3] := 1.5 ≅ (x,y,z) = (1,2,1.5)

2. ***I* = [1..*m*]×[1..*n*] (zweidimensionale Indexmenge):**

   var array *x* [1..*m*, 1..*n*]: t
   ≅ Matrix von Variablen

   $$\begin{pmatrix} x[1,1], & x[1,2], & \ldots, & x[1,n] \\ x[2,1], & x[2,2], & \ldots, & x[2,n] \\ \ldots & & & \\ x[m,1], & x[m,2], & \ldots, & x[m,n] \end{pmatrix}$$

   Beispiel:
   var *v*[1..3, 1..3]: **real** ≅ Matrix über IR³ x IR³

For-Schleife

... zur *Ablaufsteuerung* (>Komplexe Anweisungen) insbesondere bei der Verwendung von Arrays.

Seien *j*,*k* ∈ int Konstante, und sei *i* : int eine Variable.

for *i* := *j* **to** *k* **do** α **od**

entspricht (bzw. definiert durch)

i := *j*; **while** *i* ≤ *k* **do** α; *i* := *i*+1 **od**

i ist die *Laufvariable* der *for*-Schleife.

Anmerkungen:

- For-Schleife ist abweisende Schleife (sie wird u.U. gar nicht ausgeführt).
- Oft kann zusätzlich eine Schrittweite ungleich 1 angegeben werden.

Beispiele (1/14)

Die folgenden Programme MaxSum1..3 berechnen die maximale Summe aufeinanderfolgender Zahlen im **array** *x*[1..*n*] : int.

$$m = \max\{\sum_{k=i}^{j} x[k] \mid 1 \le i \le j \le n\}$$

Beispiel: *x* =

1	2	3	4	5	6	7	8	9	10
31	-41	59	26	-53	58	97	-93	-23	84

Die maximale Summe aufeinanderfolgender Zahlen ist

x[3]+..+*x*[7] = 187

Keine andere Summe hat einen größeren Wert, z.B. ist

x[1]+..+*x*[4] = 75

Erste Lösungsidee für MaxSum:
Berechnung *aller* möglichen Summen der im Array aufeinander folgenden Zahlen und Bestimmung des Maximums.

Beispiele (2/14)

```
MaxSum1: var array x[1..n] : int; var i, j,k, s,m : int;
         input x;
         m := 0;                        // Wert der leeren Summe
         for i := 1 to n do
             for j := i to n do         // i ≤ j
                 s := 0;
                 for k := i to j do
                     s := s+x[k] // s = x[i]+..+x[j]
                 od;
                 m := max(m, s)
             od
         od; output m
```

$$m = \max\{\sum_{k=i}^{j} x[k] \mid 1 \le i \le j \le n\}$$

Dieser Algorithmus ist wegen der *drei ineinander geschachtelten Schleifen* SEHR langsam.

Beispiele (3/14)

Erste Idee zur Verbesserung (MaxSum2):

⇒ Einsparen der inneren Schleife (Laufvariable k) durch Wiederverwendung bereits errechneter Summen

Statt nacheinander

k=i:	*x*[*i*]
k=i+1:	*x*[*i*] + *x*[*i* +1]
k=i+2:	*x*[*i*] + *x*[*i* +1] + *x*[*i*+2]
	. . .
k=j:	*x*[*i*] + ... + *x*[*j* -1] +*x*[*j*]

immer wieder neu zu berechnen, ist es geschickter, bereits errechnete Werte wiederzuverwenden.

Beispiele (4/14)

```
MaxSum2: var array x[1..n] : int; var i, j,s,m : int;
         input x;
         m := 0;                        // Wert der leeren Summe
         for i := 1 to n do
             s := x[i];
             m := max(m, s);
             for j := i+1 to n do       // i < j
                 s := s+x[j];           // s = x[i]+..+x[j]
                 m := max(m, s)
             od
         od;
         output m
```

Dieser Algorithmus hat nur noch *zwei ineinander geschachtelte Schleifen* und ist wesentlicher schneller als MaxSum1....

Beispiele (5/14)

.... doch lässt uns unser Streben nach Perfektion nicht ruhen: es geht noch besser! Lösung 3 kommt von M. Shamos (1977):

```
MaxSum3: var array x[1..n] : int; var i,s,m : int;
         input x;
         m := 0;
         s := 0;
         for i := 1 to n do
             s := max{0, s+x[i]};      // s = Max(x[g]+..+x[i])
             m := max(m, s)   // m = Max(x[k]+..+x[h])
         od;
         output m
```

Dieser Algorithmus hat nur noch *eine Schleife* und ist SEHR schnell ... **aber funktioniert er auch?**

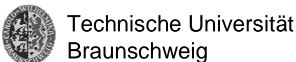
Das liegt nicht gerade auf der Hand . . .

Beispiele (6/14)

Ein Testlauf schafft Klarheit:

Beispiel: x =	1	2	3	4	5	6	7	8	9	10
	31	-41	59	26	-53	58	97	-93	-23	84
s_i	31	0	59	85	32	90	187	94	71	155
m_i	31	31	59	85	85	90	187	187	187	187

```
MaxSum3:  var array x[1..n] : int; var i,s,m : int;
          input x;
          m := 0;
          s := 0;
          for i := 1 to n do
                s := max{0, s+x[i]};
                m := max(m, s)
          od;
          output m
```



Beispiele (7/14)

Auswertung der Funktion MaxSum3

```
MaxSum3:
var array x[1..n] : int;
var i,s,m : int;
  input x;
    m := 0;
    s := 0;
    for i := 1 to n do
      s := max(0, s+x[i]);
      m := max(m, s)
    od;
  output m
```

- Signatur der Semantikfunktion: ⟦MaxSum3⟧: $int^n \rightarrow int$
- Resultat der Funktion ist die Belegung der Variablen m im Endzustand: ⟦MaxSum3⟧ $(x=(v_1,..,v_n)) = \sigma(m)$
- Endzustand σ ist lt. Definition definiert durch σ = ⟦ α ⟧ (σ_0), wobei α die Folge aller Anweisungen des Algorithmus ist.
- σ_0 ist definiert als $\sigma_0 = (x=(v_1,..,v_n), i=\bot, s=\bot, m=\bot)$ in Kurzschreibweise hier $(v^n, \bot, \bot, \bot)$
- Gesucht: MaxSum3(x)

σ = ⟦ α ⟧ (σ_0) = ⟦ α ⟧ $(v^{10}, \bot, \bot, \bot)$
 = ⟦ m := 0; s := 0; for B do $ß$ od ⟧ $(v^{10}, \bot, \bot, \bot)$
 = ⟦ s := 0; for B do $ß$ od ⟧ (⟦ m := 0 ⟧ $(v^{10}, \bot, \bot, \bot)$)

Beispiele (8/14)

```
MaxSum3:
var array x[1..n] : int;
var i,s,m : int;
  input x;
    m := 0;
    s := 0;
    for i := 1 to n do
      s := max(0, s+x[i]);
      m := max(m, s)
    od;
  output m
```

... = ⟦ s := 0; for B do $ß$ od ⟧ (⟦ m := 0 ⟧ $(v^{10}, \bot, \bot, \bot)$)
 = ⟦ s := 0; for B do $ß$ od ⟧ $(v^{10}, \bot, \bot, \bot)_{\langle m \leftarrow 0 \rangle}$
 = ⟦ s := 0; for B do $ß$ od ⟧ $(v^{10}, \bot, \bot, \mathbf{0})$
 = ⟦ for B do $ß$ od ⟧ (⟦ s := 0 ⟧ $(v^{10}, \bot, \bot, 0)$)
 = ⟦ for B do $ß$ od ⟧ $(v^{10}, \bot, 0, 0)$
 = ⟦ i := 1; while i ≤ 10 do $ß$; i:= i+1 od ⟧ $(v^{10}, \bot, 0, 0)$
 = ⟦ while B‘ do $ß$‘ od ⟧ (⟦ i := 1 ⟧ $(v^{10}, \bot, \mathbf{0}, 0)$)
 = ⟦ while B‘ do $ß$‘ od ⟧ $(v^{10}, \mathbf{1}, 0, 0)$
 = { $(v^{10}, 1, 0, 0)$, falls σ(i ≤ 10) = (1 ≤ 10) = false
 ⟦ while B‘ do β‘ od ⟧ (⟦**s:**=max(0,s+x[i]);m:=max(m,s); i**:**= i+1⟧ (σ)), sonst
 = ⟦ while B‘ do β‘ od ⟧ (⟦**s:**=max(0,s+x[i]);m:=max(m,s); i**:**= i+1⟧ $(v^{10},1,0,0)$)
 = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ (⟦**s:**=max(0,s+x[i])⟧ $(v^{10},1,0,0)$))
 = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ (⟦**s:**=31⟧ $(v^{10},1,0,0)$))

Beispiele (9/14)

```
MaxSum3:
var array x[1..n] : int;
var i,s,m : int;
  input x;
    m := 0;
    s := 0;
    for i := 1 to n do
      s := max(0, s+x[i]);
      m := max(m, s)
    od;
  output m
```

... = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ $(v^{10},1,31,0)$)
 = ⟦ while B‘ do β‘ od ⟧ (⟦**i:**= i+1⟧ (⟦m:=max(m,s)⟧ $(v^{10},1,31,0)$))
 = ⟦ while B‘ do β‘ od ⟧ (⟦**i:**= i+1⟧ $(v^{10},1,31,31)$)
 = ⟦ while B‘ do β‘ od ⟧ $(v^{10},2,31,31)$
 = { $(v^{10},2,31,31)$, falls σ(2 ≤ 10) = (2 ≤ 10) = false
 ⟦ while B‘ do β‘ od ⟧ (⟦**s:**=max(0,s+x[i]); m:=max(m,s); i**:**= i+1⟧ (σ)), sonst
 = ⟦ while B‘ do β‘ od ⟧ (⟦**s:**=max(0,s+x[i]);m:=max(m,s); i**:**= i+1⟧ $(v^{10},2,31,31)$)
 = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ (⟦**s:**=max(0,s+x[i])⟧ $(v^{10},2,31,31)$))
 = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ (⟦**s:**=0⟧ $(v^{10},2,31,31)$))
 = ⟦ while B‘ do β‘ od ⟧ (⟦m:=max(m,s); i**:**= i+1⟧ $(v^{10},2,0, 31)$)
 = ⟦ while B‘ do β‘ od ⟧ (⟦**i:**= i+1⟧ (⟦m:=max(m,s)⟧ $(v^{10},2,0,31)$))
 = ⟦ while B‘ do β‘ od ⟧ (⟦**i:**= i+1⟧ $(v^{10},2,0,31)$)
 = ⟦ while B‘ do β‘ od ⟧ $(v^{10},3,0,31)$

Beispiele (10/14)

... $= [\![$ while B' do β' od $]\!]$ (v^{10},3,0,31)

$=$ { (v^{10},3,0,31), falls σ(3 ≤ 10) = (3 ≤ 10) = false
　　$[\![$ while B' do β' od $]\!]$ ($[\![$s:=max(0,s+x[i]); m:=max(m,s); i:= i+1$]\!]$ (σ)), sonst

$= [\![$ while B' do β' od $]\!]$ ($[\![$s:=max(0,s+x[i]);m:=max(m,s); i:= i+1$]\!]$ (v^{10},3,0,31))

$= [\![$ while B' do β' od $]\!]$ (v^{10},4,59,59)

$=$ { (v^{10},4,59,59), falls σ(4 ≤ 10) = (4 ≤ 10) = false
　　$[\![$ while B' do β' od $]\!]$ ($[\![$s:=max(0,s+x[i]); m:=max(m,s); i:= i+1$]\!]$ (σ)), sonst

$= [\![$ while B' do β' od $]\!]$ ($[\![$s:=max(0,s+x[i]);m:=max(m,s); i:= i+1$]\!]$ (v^{10},4,59,59))

$= [\![$ while B' do β' od $]\!]$ (v^{10},5,85,85) = ...

$=$ { (v^{10},10,155,187), falls σ(10 ≤ 10) = (10 ≤ 10) = false
　　$[\![$ while B' do β' od $]\!]$ ($[\![$s:=max(0,s+x[i]); m:=max(m,s); i:= i+1$]\!]$ (σ)), sonst

```
MaxSum3:
var array x[1..n] : int;
var i,s,m : int;
  input x;
    m := 0;
    s := 0;
    for i := 1 to n do
      s := max(0, s+x[i]);
      m := max(m, s)
    od;
  output m
```

Beispiele (11/14)

Korrektheit des Algorithmus (1/2):

Vorüberlegung:

1. s_i *ist* die maximale Summe der Form $x[g]+..+x[i]$, $1 \leq g \leq i+1$, die an der Stelle i endet
2. für $g = i+1$ ist dies die leere Summe mit dem Wert 0, d.h. es ist immer $s_i \geq 0$.

```
MaxSum3:  var array x[1..n] : int; var i,s,m : int;
          input x;
          m := 0;
          s := 0;
          for i := 1 to n do
                s := max{0, s+x[i]};
                m := max(m, s)
          od;
          output m
```

3. Wenn Überlegung 1 (und 2) richtig ist, so muss m_n als Maximum dieser Summen s_i für $1 \leq i \leq n$ die maximale Summe aufeinanderfolgender Elemente sein

Beweis (von Überlegung 1):

Vollständige Induktion!

Exkurs: Vollständige Induktion (1/2)

Die Vollständige Induktion ist ein formales Verfahren (ein Beweisschema) zum Beweis von Aussagen über die Menge der natürlichen Zahlen IN_0

Idee (anschaulich): zeigt man, dass

wenn die Aussage für ein $n \in$ IN gilt, sie auch für $n+1$ gilt,

so gilt sie für alle $n \in$ IN

Grob mathematisch:

$A(1) \wedge (A(n) \Rightarrow A(n+1)) \Rightarrow \forall\, n \in \text{IN}: A(n)$

Exkurs: Vollständige Induktion (2/2)

Schritte der vollständigen Induktion:

0. **Formulieren der Aussage (Induktionsannahme):**
   $A(n)$ für $n \in M \subseteq IN_0$ (i.d.R. $M = IN_{(0)}$)
1. **Induktionsanfang: I.A.**
   Zeigen, dass die Aussage für ein erstes $n \in IN_0$ gilt: i.d.R. A(1)
2. **Induktionsvoraussetzung: I.V.**
   Niederschreiben von A(n)
3. **Induktionsbehauptung: I.B.**
   Niederschreiben von A(n+1)
4. **Induktionsschritt: I.S.**
   $A(n) \Rightarrow A(n+1)$ beweisen

Beispiele (12/14)

Korrektheit des Algorithmus (2/2):

Beweis der Vorüberlegung:

z.Z.: s_i *ist* die maximale Summe der Form $x[g]+..+x[i]$, $1 \leq g \leq i+1$, die an der Stelle *i* endet

I.A.: n=1: $s_n = s_1 = max\{0, s+x[n]\} = max\{0, s+x[1]\}$.

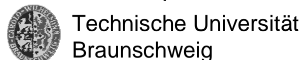
Fall 1: $x[1] \geq 0 \Rightarrow s_1 = x[1]$ = max. Summe, die an Stelle 1 endet

Fall 2: $x[1] < 0 \Rightarrow$ max. Summe, die an Stelle 1 endet, ist gleich $0 = max\{0, s_0+x[1]\} = max\{0, 0+x[1]\} = s_1$

I.V.: s_n ist die max. Summe, die an der Stelle *n* endet

I.B.: s_{n+1} ist die max. Summe, die an der Stelle *n*+1 endet

I.S.: Beweis durch Widerspruch: $s_{n+1} = max\{0, s_n+x[n+1]\}$
Annahme: s_{n+1} *ist nicht* die max. Summe, die an der Stelle *n*+1 endet $\Rightarrow \exists\ r = x[g]+..+x[n]$ für $1 \leq g \leq n$, so dass $r+x[n+1]$ ist maximale Summe, die an Stelle n+1 endet $\Rightarrow r+x[n+1] > s_n+x[n+1] \Rightarrow r > s_n$ im Widerspruch zur I.V., da s_n die max. Summe ist, die bei *n* endet. qed

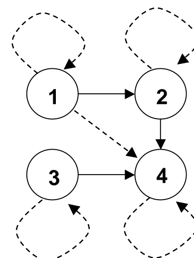


Beispiele (13/14)

Floyd-Warshall-Algorithmus

Sei $A = \{a_1, \ldots, a_n\}$ eine endliche Menge von Orten. Gegeben seien direkte Wege $a_i \rightarrow a_j$ zwischen einigen dieser Orte (Einbahnstraßen!).

Zu berechnen ist, welcher Ort von welchem Ort **erreichbar** ist, $a_i \rightarrow^* a_k$.



Adjazenzmatrizen

Eingabe

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	0	0	0	1
4	0	0	0	0

→

Ergebnis

	1	2	3	4
1	1	1	0	1
2	0	1	0	1
3	0	0	1	1
4	0	0	0	1

Mathematisch handelt es sich um die Berechnung der **reflexiven und transitiven Hülle** $\rightarrow^*$ einer Relation $\rightarrow \subseteq A \times A$.

Beispiele (14/14)

Für $1 \leq i, j \leq n$ sei $r = (r_{ij})$ die Boolesche *Adjazenzmatrix* von $\rightarrow$:

$$r_{ij} = \begin{cases} 1, \text{ falls } a_i \rightarrow a_k \\ 0 \text{ sonst} \end{cases}$$

```
FloyWars:  var array r[1..n, 1..n] : bool; var i, j,k: int;
           input s;          // die Adjazenzmatrix; initial r => s
           for i := 1 to n do s[i,i]=1 od;     // die Schlingen
           for k := 1 to n do
                for i := 1 to n do
                     for j := 1 to n do
                          s[i,j] := s[i,j]  ∨ (s[i,k] ∧ s[k,j] )
                     od
                od
           od;
           output s          // s=r* ist die Adjazenzmatrix von →*
```

Korrektheitsbeweis:
s. Goos Bd. 1, S. 59.

4.4 Prozeduren

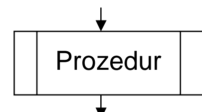
. . . dienen der Abstraktion und Wiederverwendung von Anweisungen:

- **Abstraktion** : Verbergen von Details, eine Prozedur wirkt nach außen wie eine elementare Anweisung.
- **Wiederverwendung** : Eine Prozedur wird nur einmal deklariert und kann beliebig oft verwendet *(aufgerufen)* werden.

Darstellung der Verzweigung zu einer Prozedur im Flussdiagramm:

> Notation des Aufrufs durch Nennung des Namens in Kasten mit Doppelstrich

> Jede Prozedur wird mit einem eigenen Flussdiagramm dokumentiert!



Blöcke (1/2)

1. Blöcke werden zur „Klammerung“ eines semantisch/pragmatisch abgrenzbaren Teils des „Gesamtprogramms“ (selbst Block) gebildet
2. Die Abgrenzung erfolgt durch geeignete syntaktische Elemente (i.d.R. Klammern wie z.B. {...} oder Schlüsselworte wie do...od)

Beispiel:

FloyWars:

```
var array r[1..n, 1..n] : bool; var i, j,k: int;              Block 0
input s;
for i := 1 to n do [s[i,i]=1] od;   <-- Block 1
for k := 1 to n do
    for i := 1 to n do                                         Block 2
        for j := 1 to n do                                     Block 2.1
            [ s[i,j] := s[i,j] ∨ (s[i,k] ∧ s[k,j]) ]  <-- Block 2.1.1
        od
    od
od;
output s
```

Blöcke (2/2)

3. Ein Block klammert ebenso den Kontrollfluss:
   Begonnen wird mit der *ersten* Anweisung *innerhalb des Blockes.* Gründe für die Abgabe der Kontrolle: *letzte Anweisung* des Blockes ausgeführt oder *spezifische Anweisung* zum Verlassen des Blocks (**return**, **break** etc.).
4. Blöcke werden verwendet zur Klammerung der Anweisungen
   - bei bedingter Ausführung (then...fi oder then...else...fi
   - bei Schleifen (do...od)
   - bei Prozeduren und Funktionen (s.u.)
   - etc.

   Immer dort, wo aus syntaktischen Gründen *nur eine* Anweisung stehen darf
5. In Blöcken können *lokale Variablen* deklariert werden
   Probleme:
   - Welche außerhalb des Blockes deklarierten Bezeichner (Variablen) sind innerhalb des Blockes *sichtbar bzw. gültig?*
   - Wie steht es um die *Sichtbarkeit bzw. Gültigkeit* innerhalb eines Blockes deklarierter Variablen *nach* Ausführung des Blockes?

Gültigkeitsbereich und Lebensdauer (1/3)

- bisher: alle Bezeichner für Variablen, Konstanten etc. müssen unterschiedlich sein und sind während der gesamten Programmausführung sichtbar bzw. gültig
- bei imperativen Programmiersprachen häufig: hierarchische Behandlung von Bezeichnern (>Blockstruktur)

Gültigkeitsbereich:

1. Bezeichner werden vereinbart in *Deklarationen* (> Variable, Konstante, Prozeduren und Funktionen, Typen etc.) und *formalen Parameterlisten* (von Prozeduren und Funktionen; s.u.)
2. Jedem Bezeichner kann ein *Gültigkeitsbereich* (Sichtbarkeits-bereich, scope) zugeordnet werden, in dem er in einer bestimmten Bedeutung gebraucht werden kann

Gültigkeitsbereich und Lebensdauer (2/3)

3. Bezeichner sind gültig (zunächst) in dem Block, in dem sie deklariert sind, d.h.
   - für Anweisungen aus dem Anweisungsteil dieses Blocks,
   - für Deklarationen, die auf die eigene folgen,
   - für Anweisungen aller Blöcke, die in diesen Block eingebettet sind (auch vorangehende).
4. Innerhalb eines Blocks X sind alle im umgebenden Block Y deklarierten Objekte gültig, sofern nicht im Block X ein lokales Objekt mit gleichem Namen deklariert wurde, deren Gültigkeit dann die des ersten überdeckt.
5. Vordefinierte Bezeichner (z.B. **int**, **true** etc.) sind in einem fiktiven Block deklariert, der jedes Programm umgibt (vordeklariert).

Gültigkeitsbereich und Lebensdauer (3/3)

Anmerkungen:

- In einem Block (bzw. einem Prozedurkopf; s.u.) deklarierte Variable werden als "*Lokale Variablen*" des Blockes bezeichnet. "*Globale Variable*" sind in einem umgebenden Block deklariert.
- *Gültigkeit* ist eine "statische" Eigenschaft des Programmtextes
- *Lebensdauer* (einer Variablen) ist eine "dynamische" Eigenschaft und bezeichnet den Zeitraum der Verfügbarkeit eines Wertes (einer Variablen) während der Laufzeit:
  - Bei Ausführung eines Blockes werden Speicherplätze zur Verfügung gestellt für die lokal deklarierten Variablen
  - Die zugewiesenen Werte dieser Größen sind abrufbar (d.h. sie ‚leben') bis zur Beendigung des Blockes (d.h. auch während der Ausführung enthaltener Blöcke, selbst wenn sie in diesen nicht gültig sind)
  - Sobald der Kontrollfluss den Block ‚verlässt' wird dieser Speicherplatz wieder freigegeben, d.h. Werte der lokalen Objekte und der formalen Parameter sind verloren.

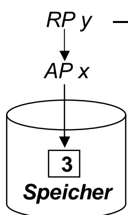
Deklaration von Prozeduren (1/3)

Deklaration: proc $P(p_1, \dots, p_n)$ $\{\{\ \delta\ ;\ \alpha(p_1, \dots, p_n)\}\}$ → ***Block***

- **Name** *P*: frei wählbar.
- **Parameter** $p_1, \dots, p_n$: lokale Variable mit eigenem Typ (formale Parameter), denen bei Aufruf der Funktion (je nach Typ) Werte, Variable, Prozeduren etc. (aktuelle Parameter) zugewiesen werden. Unterscheidung von *Werte-* und *Referenzparametern*:
  - Werteparameter (call by value)
    - Der *formale Parameter* ist lokale Variable, deren Gültigkeit auf die Ausführung der Prozedur (=Block) beschränkt ist.
    - Der formale Parameter hat keine direkte Auswirkung (keinen direkten Seiteneffekt) auf die Belegung einer Variablen des aufrufenden Programms.
    - Der *aktuelle Parameter* im Prozeduraufruf kann Konstante oder Variable sein.
    - Die Deklaration erfolgt ohne Präfix **var**.

Deklaration von Prozeduren (2/3)

- **Parameter** $p_1, \dots, p_n$ (Fortsetzung):



  - Referenzparameter (call by reference)
    - Formaler Parameter ist lokale Variable, deren Speicherplatz mit dem des aktuellen Parameters gleichgesetzt wird (es wird sozusagen anstelle eines Wertes ein Zeiger (reference) auf den Speicherplatz der Variablen im Prozeduraufruf übergeben).
    - Der aktuelle Parameter im Prozeduraufruf *muss* eine Variable sein.
    - Deklaration: Variable werden mit dem Präfix **var** gekennzeichnet.
  - Weitere bei der imperativen Programmierung bekannte Parameterarten (im Folgenden nicht weiter betrachtet):
    - *call by name* („syntaktische" Ersetzung des formalen Parameters durch den aktuellen Parameter bei Prozeduraufruf > selten) und
    - *call by value-result* (ähnlich call by reference, allerdings wird der Wert des formalen Parameters erst am Ende der Prozedurausführung dem aktuellen Parameter (Variable!) zugewiesen > ebenfalls selten)

Deklaration von Prozeduren (3/3)

Deklaration: proc $P(p_1, \dots, p_n)$ $\{\{\ \delta\ ;\ \alpha(p_1, \dots, p_n)\}\}$

- **Rumpf** {{ ... }}, in diesem:
  1. Deklarationen δ:
     lokale Konstanten, Variablen, Prozeduren . . .
  2. Anweisung $\alpha(p_1, \dots, p_n)$ (i.a. zusammengesetzt), in der die Parameter $p_1, \dots, p_n$ vorkommen.

Anmerkungen:

- Bzgl. weiterer Möglichkeiten der Parameterübergabe s. Buch bzw. Programmierkurs. Prozedurparameter behandeln wir hier nicht.
- Der Rumpf ist als Block zu interpretieren, wobei die formalen Parameter als lokale Variablen zu behandeln sind (Sonderfall Referenzparameter: der Speicherplatz des entsprechenden Variablen des aufrufenden Blockes wird nach Ausführung der Prozedur natürlich nicht freigegeben)

Beispiele (1/2)

Vertausche die Inhalte der Variablen *x* und *y* und addiere zu beiden den Wert *a* hinzu.

```
proc vertausche_plus (a : int; var x, y : int)
        // x und y sind Referenzparameter, a ist Wertparameter!
{{      var z : int;            // lokale Variable
         z := x;                // Die folgenden Veränderungen der formalen
                                // Parameter verändern die Variablen(inhalte)
                                // der als aktuelle Parameter übergebenen
                                // Variablen
         x := y;  y := z;       // Tausch der Variableninhalte
         x := x + a; y := y + a;   // Addition von a
}}
```

Die Wirkung ist die simultane Ersetzung $\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} y + a \\ x + a \end{pmatrix}$

Beispiele (2/2)

Aufrufen lässt sich die Prozedur in wechselnden Umgebungen mit wechselnden aktuellen Parametern:

```
. . . , vertausche_plus ( 0, i, j ) , . . . ,          // i, j Variable !
. . . , vertausche_plus ( 3, a[5], a[8] ) , . . . ,
. . . , vertausche_plus (a[1],a[5],a[8]) , . . . , // Parameter a:
                                                      //      Wert von a[1] ! . . . ,
. . . , vertausche_plus (i, i, j ), . . . ,            // Was kommt hierbei raus?
```

Anmerkungen:

- Als aktuelle Wertparameter erlauben wir alles, was sinnvoll ausgewertet werden kann, analog zu rechten Seiten von Zuweisungen (s.o.)

Funktionen

. . . dienen ebenfalls der Abstraktion und Wiederverwendung, aber nicht von *Anweisungen*, sondern von *Ausdrücken* (Termen).

Funktionen sind weitgehend wie Prozeduren aufgebaut. Hinzu kommt ein Term $t(p_1, \ldots, p_n)$ mit den gegebenen formalen Parametern.

Deklaration: fun $F(p_1, \ldots, p_n){:}\tau$ {{ δ , $\alpha(p_1, \ldots, p_n)$; $t(p_1, \ldots, p_n)$ }}

Die syntaktische Struktur der Funktionen (syn. Funktionsprozeduren) ist aus didaktischen Gründen vereinfacht. Dass dies für praktische Zwecke zu einschränkend ist, zeigt Beispiel 3 auf der nächsten Folie.

Auswertung von $F(p_1, \ldots, p_n){:}\tau$ mit aktuellen Parametern und Ergebnistyp τ:

- Ausführung von $\alpha(p_1, \ldots, p_n)$
- $F(p_1, \ldots, p_n) :=$ Wert von $t(p_1, \ldots, p_n){:}\tau$ am Ende der Ausführung

Beispiele (1/4)

GGT2 (s. Bsp. 6, Folie 88) als Funktion in 3 Varianten

```
fun GGT2.1 (a,b : int): int
        // a und b sind Wertparameter; Aufruf z.B. x := GGT2.1(3,9)
{{      var r,x,y : int;         // lokale Variable
         x := a; y := b; r := 1;
         while r ≠ 0 do r := x mod y; x := y; y := r od;
                          // x = ggT(a,b), sofern a,b > 0
         if a > 0 ∧ b > 0 then x fi
}}
```

Auswertung dieser Anweisung ergibt den *Rückgabewert* der Funktion (ggf. Term). In diesem Falle ist der Funktionswert ggf. undefiniert!

Anmerkung: Häufig wird nicht das Ergebnis der letzten Anweisung (bzw. des letzten Terms) des Blockes als Funktionsergebnis interpretiert, sondern die explizite Zuweisung eines Wertes zu einer Variablen verlangt, deren Bezeichner (und Typ) dem der Funktion entspricht. Ggf. wird auch eine spezielle **return**-Anweisung verwendet.

Beispiele (2/4)

```
fun GGT2.2 (var x,y : int): int
        // x und y sind Referenzparameter; Aufruf z.B. x := GGT2.2(3,9)
{{      var r : int;            // lokale Variable
        if x > 0 ∧ y > 0 then
                r := 1;
                while r ≠ 0 do r := x mod y; x := y; y := r od
        fi;
        x
}}
```

Auswertung dieses Terms ergibt den *Rückgabewert* der Funktion.

In diesem Falle ist der Funktionswert immer definiert (sofern x definiert ist)! Wird die gleiche Funktion wie bei GGT2.1 berechnet?

Offensichtlich nicht!

Seien *a* und *b* **int**-Variablen mit der Belegung -3 und 9.

GGT2.1(-3,9) = ⊥ = ggT(-3,9) ≠ -3 = GGT2.2(a,b) = a (nach Ausführung)

Beispiele (3/4)

```
fun GGT2.3 (var x,y : int): int
        // x und y sind Referenzparameter; Aufruf z.B. x := GGT2.3(3,9)
{{      var r : int;            // lokale Variable
        if x > 0 ∧ y > 0 then
                r := 1;
                while r ≠ 0 do r := x mod y; x := y; y := r od;
                x
        fi
}}
```

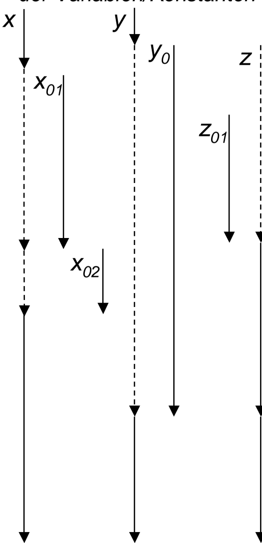
Auswertung dieses Blocks ergibt den *Rückgabewert* der Funktion.
In diesem Falle ist der Funktionswert wieder ggf. undefiniert (falls x<0 oder y<0) !

Beispiele (4/4)

```
Gültigkeit: var x : real; const y = 2.2;
            proc null (var y : real)
            {{   fun nullEins (x : real) :real
                 {{   var z : int;
                      for z := 1 to 5 do x := x+1.0 od;
                      x
                 }};
                 proc nullZwei (var x : real)
                 {{   x := x+z   }};
                 y := z+nullEins(x);
                 nullZwei(x)
            }};
            const z = 6.6;
            x := 1.1;
            null(x);
            output x,y,z;
```

const...: Deklaration einer Konstanten

Gültigkeit / Sichtbarkeit (|) der Variablen/Konstanten



Rekursion und Iteration ...

Prozeduren und Funktionen können rekursiv sein, d.h. sich selbst wieder aufrufen. Alternativ kann iterativ programmiert werden.

Wie soll man sich entscheiden?

Kennzeichnend für Iteration:

- die Wiederholungen werden explizit durch (Wdh.-)Anweisungen kontrolliert (Anzahl, Abbruchbedingung)
- jede Wiederholung wird abgeschlossen, bevor die nächste beginnt

Andere Möglichkeit (Rekursion):

- bei Ausführung des Algorithmus stößt man auf ein Teilproblem, das dieselbe Struktur hat, wie das ursprüngliche, d.h., bevor das Problem komplett gelöst ist, muss derselbe Algorithmus erneut gestartet werden
- Wichtig: wenigstens ein Teilproblem muss ohne Rückgriff auf den gleichen Algorithmus zu lösen sein.

Rekursive Probleme

Rekursion ist günstig bei Problemen, deren Lösung eine rekursive Struktur hat:

- Überprüfen der Syntaxdefinition eines Ausdrucks
- Operationen auf rekursiv definierten Datenstrukturen (z.B. Bäume; s.u.)
- Berechnung rekursiv definierter mathematischer Funktionen (z.B. Fakultätsfunktion, Binomialkoeffizienten etc.; s.o.)
- etc.

Allgemeines Rekursionsschema (Rückgriff auf bereits definierte Funktionswerte; Anfangswert muss ohne Rückgriff definiert sein):

Definition: X beliebige Menge, $f: IN_0 \times X \rightarrow IR$ heißt rekursiv definiert, falls es Funktionen $g: X \rightarrow IR$ und $h: IN \times X \times IR \rightarrow IR$ gibt, so dass

$f(0,x) := g(x)$ für $x \in X$

$f(n+1,x) := h(n+1,x,f(n,x))$, $x \in X$, $n \in IN$

Beispiel (1/4)

Beispiel: Fakultätsfunktion

```
fun FAK (n : int): int
{{      var result: int;          // lokale Variable für das Ergebnis
        if n = 0  then  result := 1;
                  else  result := FAK(n-1)*n
        fi;
        result
}}
```

- Damit eine rekursive Funktion einen Wert liefern kann, muss *eine Fallunterscheidung* enthalten sein, in der *mindestens ein Zweig* keinen rekursiven Aufruf enthält.
- Die Bedingung muss im Laufe der Rekursion zu **true** ausgewertet werden (ansonsten kommt es zu einer Endlosschleife)

Beispiel (2/4)

```
fun FAK (n : int): int
{{      var result: int;
        if n = 0  then  result := 1;
                  else  result := FAK(n-1)*n
        fi;
        result
}}
```

Ausführung von FAK(2) auf einem (idealisierten) Rechner?

Aufruf FAK(2)

1. Anlegen: lokale Variable n mit Wert 2 und Variable result (zunächst undefiniert)
2. Ausführen: if...then...else *result* := FAK(*n-1*)∗n

   Unterbrechen:

   Speichern - der aktuell bearbeiteten Stelle

   - Zustand (Variablenbelegungen)

   Aufruf FAK(1)

   1. Anlegen: lokale Variable *n* mit Wert 1 und Variable *result* (zunächst undefiniert)

Beispiel (3/4)

Aufruf FAK(2) ...

Aufruf FAK(1) (Fortsetzung)

2. Ausführen: if...then...else *result* := FAK(*n-1*)∗n

   Unterbrechen:

   Speichern - der aktuell bearbeiteten Stelle

   - Zustand (Variablenbelegungen)

   Aufruf FAK(0)

   1. Anlegen: lokale Variable *n* mit Wert 0 und Variable *result* (zunächst undefiniert)
   2. Ausführen: if...then *result* := 1; Rückgabe result=1

   Fortsetzung FAK(1) an unterbrochener Stelle mit gespeichertem Zustand und Returnwert : Rückgabe result=1

Fortsetzung FAK(2) an unterbrochener Stelle mit gespeichertem Zustand und Returnwert : Rückgabe result=2; ENDE

Beispiel (4/4)

> Rekursive Lösung erzeugt pro Rekursionsschritt einen nicht zu unterschätzenden Speicherbedarf, der im Moment des letzten Rekursionsschrittes seinen Höhepunkt erreicht.

Iterative Lösung der Fakultätsfunktion

```
fun FAK (n : int): int
{{     var z, result: int;
       result := 1;
       for z := 2 to n do result := result ∗ z od;
       result
}}
```

> Iterative Lösung hat unabhängig vom Parameter *n* einen festen Speicherbedarf.

... Rekursion und Iteration

Wie soll man sich entscheiden?

- Hat das Problem eine rekursive Struktur, so kann die Lösung (Prozedur oder Funktion) meist elegant (kurz) rekursiv formuliert werden.
- Vorteil der rekursiven Formulierung (u.a.): Ersparnis expliziter Hilfsvariablen (Laufindex) und Kontrollstrukturen.
- Der gesparte Aufwand muss zur Laufzeit erbracht werden (vom Laufzeitsystem/der Programmumgebung). Dabei ist meist mehr Speicherplatz nötig, als bei nicht-rekursiven Lösungen.
- Das korrekte Arbeiten rekursiver Funktionen/Prozeduren ist oft schwierig nachzuweisen.

Empfehlung:

Rekursive Prozeduren da anwenden, wo sie durch die Problemstruktur nahegelegt werden.