

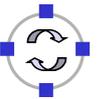
Programmierung und Algorithmen

Kapitel 4

Algorithmenparadigmen

P&A (WS 22/23): 04 – Algorithmenparadigmen

1



Überblick

Einführung

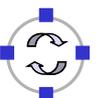
Applikative Algorithmen

Imperative Algorithmen

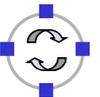
Deduktive Algorithmen

P&A (WS 22/23): 04 – Algorithmenparadigmen

2



- ❑ Begriff **Algorithmenparadigma**
 - ❑ Allgemein: „*Paradigma*“ [griechisch] das: *Beispiel, Muster*. (Brockhaus)
 - ❑ Informatik: „Muster“ für Entwurf und Formulierung von Algorithmen
- ❑ Grundlegende Arten zur Notation von Algorithmen
 - ❑ **Applikativ**: Verallgemeinerung der Funktionsauswertung
 - ❑ **Imperativ**: basierend auf einem einfachen Maschinenmodell mit gespeicherten und änderbaren Werten
 - ❑ **Deduktiv**: basierend auf logischen Aussagen und Schlussfolgerungen
 - ❑ Weitere Paradigmen (objektorientiert, ...)
- ❑ Java: objektorientiert, imperativ, Elemente von applikativ



Grundidee:

- ❑ Definition zusammengesetzter Funktionen durch Terme:

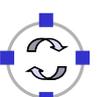
$$f(x) = 5x + 1$$

- ❑ Unbestimmte:
 - ❑ x, y, z, \dots vom Typ `int`
 - ❑ q, p, r, \dots vom Typ `bool`
- ❑ Terme mit Unbestimmten:
 - ❑ Terme vom Typ `int`

$$x, x - 2, 2x + 1, (x + 1)(y - 1)$$

- ❑ Terme vom Typ `bool`

$$p, p \wedge \text{true}, (p \vee \text{true}) \Rightarrow (q \vee \text{false})$$

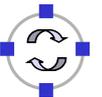


Sind v_1, \dots, v_n Unbestimmte vom Typ T_1, \dots, T_n (**bool** oder **int**) und ist $t(v_1, \dots, v_n)$ ein Term vom Typ T , so heißt

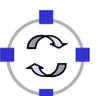
$$f(v_1, \dots, v_n) = t(v_1, \dots, v_n)$$

eine **Funktionsdefinition** vom Typ T .

- f : Funktionsname
- v_1, \dots, v_n : formale Parameter
- $t(v_1, \dots, v_n)$: Funktionsausdruck
- Zur Definition von Termen siehe Kapitel 2, Folie 48 zu int-Termen



1. $f(p, q, x, y) = \text{if } p \vee q \text{ then } 2x + 1 \text{ else } 3y - 1 \text{ fi}$
2. $g(x) = \text{if even}(x) \text{ then } x \div 2 \text{ else } 3x - 1 \text{ fi}$
3. $h(p, q) = \text{if } p \text{ then } q \text{ else false fi}$



1. $f(p, q, x, y) = \text{if } p \vee q \text{ then } 2x + 1 \text{ else } 3y - 1 \text{ fi}$

$f: \text{bool} \times \text{bool} \times \text{int} \times \text{int} \rightarrow \text{int}$

$f(\text{true}, \text{true}, 3, 4) = 7$

2. $g(x) = \text{if even}(x) \text{ then } x \div 2 \text{ else } 3x - 1 \text{ fi}$

$g: \text{int} \rightarrow \text{int}$

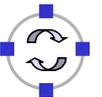
$g(2) = 1, g(3) = 8$

3. $h(p, q) = \text{if } p \text{ then } q \text{ else false fi}$

$h: \text{bool} \times \text{bool} \rightarrow \text{bool}$

$h(\text{false}, \text{false}) = \text{false}$

Bemerkung: $h(p, q) = p \wedge q$



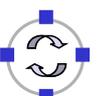
- Erweiterung der Definition von Termen
- Neu: **Aufrufe** definierter Funktionen sind Terme
- Beispiel für Einsatz in Funktionsdefinitionen:

$f(x, y) = \text{if } g(x, y) \text{ then } h(x + y) \text{ else } h(x - y) \text{ fi}$

$g(x, y) = (x = y) \vee \text{odd}(y)$

$h(x) = j(x + 1) \cdot j(x - 1)$

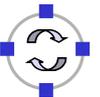
$j(x) = 2x - 3$



```

f(1, 2) ⇨   if g(1, 2) then h(1 + 2) else h(1 - 2) fi
            ⇨   if 1 = 2 V odd(2) then h(1 + 2) else h(1 - 2) fi
            ⇨   if false then h(1 + 2) else h(1 - 2) fi
            ⇨   h(1 - 2)
            ⇨   h(-1)
            ⇨   j(-1 + 1) · j(-1 - 1)
            ⇨   j(0) · j(-1 - 1)
            ⇨   j(0) · j(-2)
            ⇨   (2 · 0 - 3) · j(-2)
            ⇨*  (-3) · (-7)
            ⇨   21

```



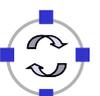
$$f(x, y) = \text{if } x = 0 \text{ then } y \\ \text{else if } x > 0 \text{ then } f(x - 1, y) + 1 \\ \text{else } -f(-x, -y) \text{ fi fi}$$

Auswertungen

```

f(0, y)      ⇨ y für alle y
f(1, y)      ⇨ f(0, y) + 1 ⇨ y + 1
f(2, y)      ⇨ f(1, y) + 1 ⇨ (y + 1) + 1 ⇨ y + 2
...
f(n, y)      ⇨ n + y für alle n ∈ int, n > 0
f(-1, y)     ⇨ -f(1, -y) ⇨ -(1 - y) ⇨ y - 1
...
f(x, y)      ⇨ x + y für alle x, y ∈ int

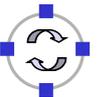
```



Ein **applikativer Algorithmus** ist eine Menge von Funktionsdefinitionen

$$\begin{aligned} f_1(v_{1,1}, \dots, v_{1,n_1}) &= t_1(v_{1,1}, \dots, v_{1,n_1}), \\ &\vdots \\ f_m(v_{m,1}, \dots, v_{m,n_m}) &= t_m(v_{m,1}, \dots, v_{m,n_m}). \end{aligned}$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und bestimmt die „Bedeutung“ (Semantik) des Algorithmus.



$$f(x) = \text{if } x = 0 \text{ then } 0 \text{ else } f(x - 1) \text{ fi}$$

□ Auswertungen:

$$f(0) \mapsto 0$$

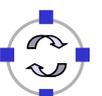
$$f(1) \mapsto f(0) \mapsto 0$$

$$f(x) \mapsto 0 \text{ für alle } x \geq 0$$

$$f(-1) \mapsto f(-2) \mapsto \dots \text{ Auswertung terminiert nicht!}$$

□ Also gilt

$$f(x) = \begin{cases} 0 & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$



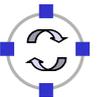
$$x! = x(x - 1)(x - 2) \dots 2 \cdot 1 \text{ für } x > 0$$

- Bekannte Definition: $0! = 1$, $x! = x \cdot (x - 1)!$
- Problem: negative Eingabewerte
- 1. Lösung:

```
fac(x) = if x = 0 then 1 else x · fac(x - 1) fi
```

- Bedeutung:

$$\text{fac}(x) = \begin{cases} x! & \text{Falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$



- 2. Lösung:

```
fac(x) = if x ≤ 0 then 1 else x · fac(x - 1) fi
```

- Bedeutung:

$$\text{fac}(x) = \begin{cases} x! & \text{Falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

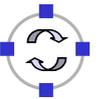
- Nachteil dieser Lösung: mathematisch nicht korrekt bei negativen Werten
- Vorteil: Algorithmus terminiert immer und liefert korrektes Ergebnis bei Einhalten des Definitionsbereiches



- Zahlenreihe
- Progression bei Vermehrung von Kaninchen



1. Am Anfang gibt es ein Kaninchenpaar.
2. Jedes Paar wird im zweiten Monat zeugungsfähig.
3. Danach zeugt es jeden Monat ein weiteres Paar.
4. Kein Kaninchen stirbt.



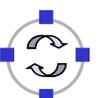
$$f_0 = f_1 = 1, \quad f_i = f_{i-1} + f_{i-2} \text{ für } i > 0$$

- Dabei ist f_{i-1} die Anzahl der Kaninchenpaare im letzten Monat und f_{i-2} die Anzahl der Kaninchenpaare im vorletzten Monat (i bezeichnet dabei den aktuellen Monat)

$$fib(x) = \begin{cases} 1 & \text{if } (x = 0) \vee (x = 1) \\ fib(x - 2) + fib(x - 1) & \text{fi} \end{cases}$$

- Bedeutung:

{	x -te Fibonacci-Zahl	falls $x > 0$
	\perp	falls $x < 0$
	1	sonst



- Ausnutzung folgender Regeln:

$$0 \cdot y = 0$$

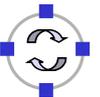
$$x \cdot y = (x - 1) \cdot y + y \quad \text{für } x > 0$$

$$x \cdot y = -((-x) \cdot y) \quad \text{für } x < 0$$

```

prod(x, y) =  if x = 0    then 0 else
              if x > 0    then prod(x - 1, y) + y
              else - prod(- x, y) fi fi

```



- Für $0 < x$ und $0 < y$ gilt:

$$ggT(x, x) = x$$

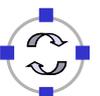
$$ggT(x, y) = ggT(y, x)$$

$$ggT(x, y) = ggT(x, y - x) \quad \text{für } x < y$$

```

ggT(x, y) =  if (x ≤ 0) ∨ (y ≤ 0)
              then ggT(x, y)
              else if x = y
                    then x
                    else if x > y
                          then ggT(y, x)
                          else ggT(x, y - x) fi fi fi

```



- ggT korrekt für positive Eingaben, bei negativen Eingaben undefiniert (Berechnung terminiert nicht – der ggT kann hierfür jedoch auch sinnvoll definiert werden)

$$\begin{array}{lll}
 ggT(39, 15) & \mapsto ggT(15, 39) & \mapsto ggT(15, 24) \\
 & \mapsto ggT(15, 9) & \mapsto ggT(9, 15) \\
 & \mapsto ggT(9, 6) & \mapsto ggT(6, 9) \\
 & \mapsto ggT(6, 3) & \mapsto ggT(3, 6) \\
 & \mapsto ggT(3, 3) & \mapsto 3
 \end{array}$$

- Berechnungsschema ist Formalisierung des Originalverfahrens von Euklid
(Euklid: Elemente, 7. Buch, Satz 2; ca 300 v.Chr.).



```

even(0)      = true
odd(0)       = false
even(x + 1) = odd(x)
odd(x + 1)  = even(x)

```

```

even(x)      = if x = 0 then true else
                if x > 0 then odd(x - 1)
                else odd(x + 1) fi fi
odd(x)       = if x = 0 then false else
                if x > 0 then even(x - 1)
                else even(x + 1) fi fi

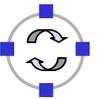
```

Algorithmus für `odd` durch Vertauschen der Reihenfolge der Funktionsdefinitionen

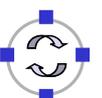


```
prim(x) =  if x ≤ 1 then false
           else pr (2, x) fi
pr (x, y) = if x ≥ y then true else
           else (y mod x) ≠ 0
              ∧ pr (x + 1, y) fi
```

- Hilfsfunktion $pr(x, y)$ ist wahr, genau dann wenn y durch keine Zahl z , $x \leq z < y$, teilbar ist
- $prim(x) \Leftrightarrow (x > 1) \wedge pr(2, x)$



- Lösung nicht sehr effizient. . .
- Verbesserungen:
 - Ersetze $x \geq y$ durch $x \cdot x \geq y$ (kleinster Teiler muss kleiner oder gleich als Quadratwurzel sein!).
 - Ersetze $pr(x + 1, y)$ durch
 $if\ x = 2\ then\ pr(3, y)\ else\ pr(x + 2, y)\ fi.$



- McCarthys 91-Funktion

$$f(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } f(f(x + 11)) \text{ fi}$$

- Beispielberechnungen:

$$f(100) \mapsto f(f(111)) \mapsto f(101) \mapsto 91$$

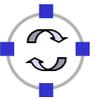
$$f(99) \mapsto f(f(110)) \mapsto f(100) \mapsto \dots \mapsto 91$$

$$\dots \mapsto \dots \mapsto 91$$

- Gilt die folgende Äquivalenz?

$$f(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } 91 \text{ fi}$$

- **Ja!** — *Beweis später in der Vorlesung.*



$$f(x) = \text{if } x = 1 \text{ then } 1 \text{ else } f(g(x)) \text{ fi}$$

$$g(x) = \text{if even}(x) \text{ then } x \div 2 \text{ else } 3x + 1 \text{ fi}$$

- Für negative Zahlen und 0:

$$f(x) = \perp \text{ für } x \leq 0$$

- Für positive Zahlen ist als definierter Wert nur 1 möglich

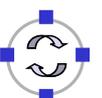
$$f(1) \mapsto 1$$

$$f(2) \mapsto f(1) \mapsto 1$$

$$f(3) \mapsto f(10) \mapsto f(5) \mapsto f(16) \mapsto f(8) \mapsto f(4) \mapsto f(2) \mapsto 1$$

$$f(4) \mapsto \dots \mapsto 1$$

...



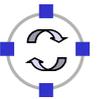
$$f(x) = \text{if } x = 1 \text{ then } 1 \text{ else } f(g(x)) \text{ fi}$$

$$g(x) = \text{if even}(x) \text{ then } x \div 2 \text{ else } 3x + 1 \text{ fi}$$

Verdacht: Es gilt folgende Äquivalenz:

$$f(x) = \text{if } x > 0 \text{ then } 1 \text{ else } \perp \text{ fi}$$

... aber noch unbewiesen.



Ackermann-Funktion

$$f(x, y) = \begin{array}{l} \text{if } x \leq 0 \text{ then } y + 1 \text{ else} \\ \quad \text{if } y \leq 0 \text{ then } f(x - 1, 1) \\ \quad \text{else } f(x - 1, f(x, y - 1)) \text{ fi fi} \end{array}$$

□ Werte wachsen unglaublich schnell an:

$$f(0, 0) \mapsto 1$$

$$f(1, 0) \mapsto f(0, 1) \mapsto 2$$

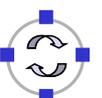
$$f(1, 1) \mapsto f(0, f(1, 0)) \mapsto f(0, f(0, 1)) \mapsto f(0, 2) \mapsto 3$$

$$f(2, 2) \mapsto f(1, f(2, 1)) \mapsto f(1, f(1, f(2, 0))) \mapsto f(1, f(1, f(1, 1)))$$

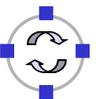
$$\dots \mapsto f(1, 5) \mapsto \dots f(0, 6) \mapsto 7$$

$$f(3, 3) \mapsto 61$$

$$f(4, 4) \mapsto 2^{2^{2^{2^{2^2}}}} - 3 = 2^{2^{65536}} - 3$$



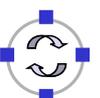
- Verbreitetste Art, Algorithmen für Computer zu formulieren
- Basiert auf den Konzepten Anweisung und Variablen
- Wird durch Programmiersprachen Java, C, PASCAL, FORTRAN, COBOL, . . . realisiert
- Hier zunächst nur Darstellung der Grundprinzipien



- Eine **Variable** besteht aus einem Namen (z. B. X) und einem veränderlichen **Wert**.
- Ist t ein Term ohne Unbestimmte und $w(t)$ sein Wert, dann heißt das Paar $x := t$ eine **Wertzuweisung**. Ihre Bedeutung ist festgelegt durch
Nach Ausführung von $x := t$ gilt $x = w(t)$.

Vor der Ausführung der ersten Wertzuweisung gilt $x = \perp$ (undefiniert).

- Beispiele:

$$X := 7$$
$$F := true$$
$$X := (3 - 7) \cdot 9$$
$$Q := \neg (true \vee \neg false) \vee \neg \neg true$$


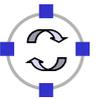
- Ist $\underline{X} = \{ X_1, X_2, \dots \}$ eine Menge von Variablen(-namen), von denen jede Werte aus der Wertemenge W haben kann (alle Variablen vom gleichen Typ), dann ist der Zustand Z eine partielle Abbildung

$Z : \underline{X} \rightarrow W$ (Zuordnung des momentanen Wertes)

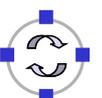
- Ist $Z : \underline{X} \rightarrow W$ Zustand und wählt man eine Variable X und einen Wert w , so ist der **transformierte Zustand** wie folgt definiert:

$$Z \langle X \leftarrow w \rangle : \underline{X} \rightarrow W \text{ mit}$$

$$Y \mapsto \begin{cases} w, & \text{falls } X = Y \\ Z(Y), & \text{sonst} \end{cases}$$



- **Ausdrücke** entsprechen im Wesentlichen den Termen einer applikativen Sprache, jedoch stehen an der Stelle von Unbestimmten nun Variablen
- Auswertung von Termen ist zustandsabhängig:
 - Gegebener Term $2X + 1$
 - Wert im Zustand Z ist $2 \cdot Z(X) + 1$



- Wert eines Ausdrucks $t(X_1, \dots, X_n)$ wird mit $Z(t(X_1, \dots, X_n))$ bezeichnet:

$$Z(2 \cdot X + 1) = 2 \cdot Z(X) + 1$$

- Damit auch möglich:

$$X := t(X_1, \dots, X_n)$$

- Transformierter Zustand ist

$$\llbracket X := t(X_1, \dots, X_n) \rrbracket (Z) = Z \langle X \leftarrow Z(t(X_1, \dots, X_n)) \rangle$$

- “Semantikkammern”: **Bedeutung einer Anweisung als Funktion auf Zuständen**



Beispielszuweisung:

$$\alpha_1 = (X := 2 \cdot Y + 1)$$

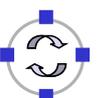
- Transformation in Funktion auf Zuständen

$$\llbracket \alpha_1 \rrbracket (Z) = Z \langle X \leftarrow 2Z(Y)+1 \rangle$$

- Zuweisung berechnet neuen Zustand

- Alter Zustand Z

Neuer Zustand $\llbracket \alpha_1 \rrbracket (Z)$



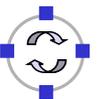
Beispielzuweisung mit der selben Variable auf beiden Seiten:

$$\alpha_2 = (X := 2 \cdot X + 1)$$

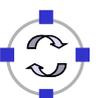
- Transformation in

$$\llbracket \alpha_2 \rrbracket (Z) = Z \langle X \leftarrow 2Z(X)+1 \rangle$$

- Bei der letzten Anweisung handelt es sich *nicht* um eine rekursive Gleichung!



- Bisher: Anweisungen als Funktionen auf Zuständen
- Konstrukte zum Aufbau von imperativen Algorithmen
 1. Sequenz
 2. Auswahl
 3. Iteration
- Semantik wird wiederum durch Konstruktion von Funktionen festgelegt



- Folge (Sequenz): Sind α_1 und α_2 Anweisungen, so ist

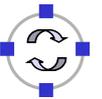
$$\alpha_1; \alpha_2$$

auch eine Anweisung.

- Zustandstransformation

$$\llbracket \alpha_1; \alpha_2 \rrbracket (Z) = \llbracket \alpha_2 \rrbracket (\llbracket \alpha_1 \rrbracket (Z))$$

- Semantik: Schachteln der Funktionsaufrufe



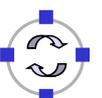
- Auswahl (Selektion): Sind α_1 und α_2 Anweisungen und B ein Boolescher Ausdruck, so ist

$$\mathbf{if\ } B \mathbf{\ then\ } \alpha_1 \mathbf{\ else\ } \alpha_2 \mathbf{\ fi}$$

eine Anweisung.

$$\llbracket \mathbf{if\ } B \mathbf{\ then\ } \alpha_1 \mathbf{\ else\ } \alpha_2 \mathbf{\ fi} \rrbracket (Z) = \begin{cases} \llbracket \alpha_1 \rrbracket (Z) & \text{falls } Z(B) = \mathbf{true} \\ \llbracket \alpha_2 \rrbracket (Z) & \text{falls } Z(B) = \mathbf{false} \end{cases}$$

- Voraussetzung: $Z(B)$ definiert — sonst Bedeutung der Auswahlanweisung undefiniert.



- Wiederholung (Iteration): Ist Anweisung und B Boolescher Ausdruck, so ist

```
while B do  $\alpha$  od
```

eine Anweisung.

$$\llbracket \text{while } B \text{ do } \alpha \text{ od} \rrbracket (Z) = \begin{cases} Z, & \text{falls } Z(B) = \text{false} \\ \llbracket \text{while } B \text{ do } \alpha \text{ od} \rrbracket (\llbracket \alpha \rrbracket (Z)), & \text{sonst} \end{cases}$$

- Ist $Z(B)$ undefiniert, so ist die Bedeutung dieser Anweisung ebenfalls undefiniert.



- In realen imperativen Programmiersprachen gibt es fast immer diese Anweisungen, meistens jedoch viel mehr.
- **while**-Schleifen sind rekursiv definiert, sie brauchen nicht zu terminieren.
- Die Verwendung von **if-fi** ist wegen der klaren Klammerung sauberer.
- Bereits Programmiersprachen mit diesen Sprachelementen sind *universell* (d.h. man kann alle berechenbaren Funktionen mit ihnen berechnen; Beweise werden in Vorlesungen zur Theorie behandelt).
- Wir beschränken uns im Folgenden auf die Datentypen **bool** und **int**.



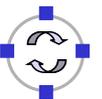
< Programmname >:

var $X, Y, \dots : \mathbf{int}; P, Q, \dots : \mathbf{bool};$ \Leftarrow Variablen-Deklaration

input $X_1, \dots, X_n;$ \Leftarrow Eingabe-Variablen

$\alpha;$ \Leftarrow (Anweisung(en))

output $Y_1, \dots, Y_m.$ \Leftarrow (Ausgabe-Variablen)

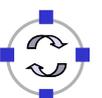


- Die Bedeutung (Semantik) eines imperativen Algorithmus ist eine partielle Funktion:

$$\begin{aligned} \llbracket \text{PROG} \rrbracket &: W_1 \times \dots \times W_n \mapsto V_1 \times \dots \times V_m \\ \llbracket \text{PROG} \rrbracket (w_1, \dots, w_n) &= (Z(Y_1), \dots, Z(Y_m)) \\ \text{wobei } Z &= \llbracket \alpha \rrbracket (Z_0), \\ Z_0(X_i) &= w_i, i = 1, \dots, n, \\ \text{und } Z_0(Y) &= \perp \text{ f\u00fcr alle Variablen } Y \neq X_i, \\ & \quad i = 1, \dots, n \end{aligned}$$

wobei gilt

PROG	Programmname
W_1, \dots, W_n	Wertebereiche der Typen von X_1, \dots, X_n
V_1, \dots, V_m	Wertebereiche der Typen von Y_1, \dots, Y_m

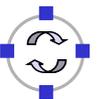


Die Algorithmenausführung imperativer Algorithmen besteht aus einer Folge von Basisschritten, genauer von Wertzuweisungen.

Diese Folge wird mittels Selektion und Iteration basierend auf booleschen Tests über dem Zustand konstruiert.

Jeder Basisschritt definiert eine elementare Transformation des Zustands.

Die Semantik des Algorithmus ist durch die Kombination all dieser Zustandstransformationen zu einer Gesamttransformation festgelegt.

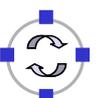


- Fakultätfunktion $0! = 1$, $x! = x \cdot (x - 1)!$ für $x > 0$

```
FAC:var X,Y: int;  
    input X;  
    Y:=1;  
    while X>1 do Y:=Y · X; X:=X-1 od  
    output Y.
```

Hier ist $\llbracket FAC \rrbracket (x) = \begin{cases} x! & \text{für } x \geq 0 \\ 1 & \text{sonst} \end{cases}$

- Falls Bedingung für die `while`-Schleife „ $X \neq 0$ “:

$$\llbracket FAC \rrbracket (x) = \begin{cases} x! & \text{für } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$


- Gesucht ist das Ergebnis des Aufrufs $FAC(3)$

- Abkürzung **while** β für die Zeile

```
while X>1 do Y:=Y * X; X:=X-1 od
```

- Signatur der Semantikfunktion:

$$\llbracket FAC \rrbracket : \text{int} \rightarrow \text{int}$$

- Funktion ist durch Lesen von Y im Endzustand Z definiert:

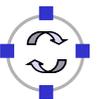
$$\llbracket FAC \rrbracket (w) = Z(Y)$$

- Endzustand Z ist definiert durch folgende Gleichung:

$$Z = \llbracket \alpha \rrbracket (Z_0)$$

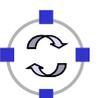
wobei α die Folge aller Anweisungen des Algorithmus ist.

- Initialer Zustand Z_0 definiert als $Z_0 = (X = w, Y = \perp)$.
- Zustände abkürzend ohne Variablennamen: $Z_0 = (w, \perp)$.

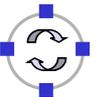


$$\begin{aligned}
 Z &= \llbracket \alpha \rrbracket (Z_0) \\
 &= \llbracket \alpha \rrbracket (3, \perp) \\
 &= \llbracket Y := 1; \text{while } \beta \rrbracket (3, \perp) \\
 &= \llbracket \text{while } \beta \rrbracket (\llbracket Y := 1 \rrbracket (3, \perp)) \\
 &= \llbracket \text{while } \beta \rrbracket ((3, \perp) \langle Y \leftarrow 1 \rangle) \\
 &= \llbracket \text{while } \beta \rrbracket (3, 1) \\
 &= \begin{cases} Z, \text{ falls } Z(B) = \text{false} \\ \llbracket \text{while } B \text{ do } \alpha' \text{ od} \rrbracket (\llbracket \alpha' \rrbracket (Z)), \text{ sonst} \end{cases} \\
 &= \begin{cases} (3, 1), \text{ falls } Z(X > 1) = (3 > 1) = \text{false} \\ \llbracket \text{while } \beta \rrbracket (\llbracket Y := Y * X; X := X - 1 \rrbracket (Z)), \text{ sonst} \end{cases} \\
 &= \llbracket \text{while } \beta \rrbracket (\llbracket Y := Y * X; X := X - 1 \rrbracket (3, 1)) \\
 &= \llbracket \text{while } \beta \rrbracket (\llbracket X := X - 1 \rrbracket (\llbracket Y := Y * X \rrbracket (3, 1))) \\
 &= \llbracket \text{while } \beta \rrbracket (\llbracket X := X - 1 \rrbracket (3, 3)) \\
 &= \llbracket \text{while } \beta \rrbracket (2, 3)
 \end{aligned}$$

... wird fortgesetzt...



$$\begin{aligned}
Z &= \llbracket \alpha \rrbracket (Z_0) \\
&\dots \\
&= \llbracket \text{while } \beta \rrbracket (2, 3) \\
&= \begin{cases} (2, 3), \text{ falls } Z(X > 1) = (2 > 1) = \mathbf{false} \\ \llbracket \text{while } \beta \rrbracket (\llbracket Y := Y * X; X := X - 1 \rrbracket (Z)), \text{ sonst} \end{cases} \\
&= \llbracket \text{while } \beta \rrbracket (\llbracket Y := Y * X; X := X - 1 \rrbracket (2, 3)) \\
&= \llbracket \text{while } \beta \rrbracket (\llbracket X := X - 1 \rrbracket (\llbracket Y := Y * X \rrbracket (2, 3))) \\
&= \llbracket \text{while } \beta \rrbracket (\llbracket X := X - 1 \rrbracket (2, 6)) \\
&= \llbracket \text{while } \beta \rrbracket (1, 6) \\
&= \begin{cases} (1, 6), \text{ falls } Z(X > 1) = (1 > 1) = \mathbf{false} \\ \llbracket \text{while } \beta \rrbracket (\llbracket Y := Y * X; X := X - 1 \rrbracket (Z)), \text{ sonst} \end{cases} \\
&= (1, 6)
\end{aligned}$$



- Der Übergang von der 3. auf die 4. Zeile folgt der Definition der Sequenz, indem der Sequenzoperator in einen geschachtelten Funktionsaufruf umgesetzt wird.
- Nur in der 5. Zeile wurde eine Wertzuweisung formal umgesetzt, später sind sie einfach verkürzt direkt ausgerechnet.
- In der 7. Zeile haben wir die Originaldefinition der Iteration eingesetzt (nur mit dem Kürzel α' statt α , da α bereits verwendet wurde). Im Beispiel gilt

$$\alpha' = \{ Y := Y * X; X : X - 1 \}$$

- Das Z in der 7. und 8. Zeile steht für den Zustand (3, 1) (in späteren Zeilen analog für den jeweils aktuellen Zustand).
- Bei diesem Beispiel sieht man Folgendes sehr deutlich: *Die Ausführung einer While-Schleife erfolgt analog einer rekursiven Funktionsdefinition!*

Damit gilt:

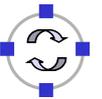
$$\llbracket \text{FAC} \rrbracket (3) = Z(Y) = (X = 1, Y = 6)(Y) = 6$$



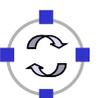
```
FIB: var X,A,B,C: int;
      input X;
      A:=1, B:=1;
      while X>0 do
          C:=A+B; A:=B; B:=C; X:=X-1
      od
      output A.
```

$$\llbracket FIB \rrbracket (x) = \begin{cases} \text{die } x\text{-te Fibonacci-Zahl,} & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

Hinweis: Die Zwischenspeicherung der beiden letzten Fibonacci-Zahlen in den Variablen A und B ist effizienter als der doppelt rekursive Aufruf der applikativen Version



```
GGT1   var X,Y: int;
        input X,Y;
        while X ≠ Y do
            while X>Y do X:=X-Y od
            while X<Y do Y:=Y-X od
        od
        output X.
```



X	Y
19	5
14	5
9	5
4	5
4	1
3	1
2	1
1	1

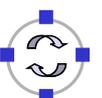


```
GGT2 var X, Y, R: int;
      input X, Y;
      R:=1;
      while R ≠ 0 do
          R:=X mod Y; X:=Y; Y:=R od
      output X.
```

Einige Berechnungen für GGT2:

X	Y	R	X	Y	R
19	5	1	2	100	1
5	4	4	100	2	1
4	1	1	2	0	0
1	0	0			

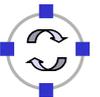
ist $X < Y$, so wird erst vertauscht: $X \leftrightarrow Y$



- Wie ist das Verhalten von GGT2 für negative X oder Y?

$$\llbracket \text{GGT2} \rrbracket (x, y) = \begin{cases} \text{ggT} (x, y), & \text{falls } x, y > 0 \\ y, & \text{falls } x = y \neq 0 \text{ oder} \\ & x = 0, y \neq 0 \\ \perp & \text{falls } y = 0 \\ \text{ggT} (|x|, |y|), & \text{falls } x < 0 \text{ und } y > 0 \\ -\text{ggT} (|x|, |y|), & \text{falls } y < 0 \end{cases}$$

- Ist GGT2 effizienter als Version 1?



- Basierend auf logischen Aussagen
- Programmiersprachen wie PROLOG
- Logische Aussagen ergeben - streng genommen - allein kein Berechnungsmodell
- Deduktiver Algorithmus =
 - Logische Aussagen +
 - Interpretationsalgorithmus +
 - Anfrage

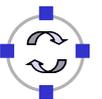


- Aussage:
Susi ist Tochter von Petra
- Aussageform mit Unbestimmten:
X ist Tochter von Y
- Belegung:

$X \mapsto \text{Susi}; Y \mapsto \text{Petra}$

- Atomare Formel:

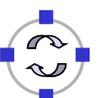
$\text{Tochter}(X, Y)$



- Alphabet
 - Unbestimmte X, Y, Z, \dots
 - Konstanten a, b, c, \dots
 - Prädikatensymbole P, Q, R, \dots mit Stelligkeit (analog der Signatur von ADT)
 - Logische Konnektive \wedge und \Rightarrow
- Atomare Formeln: $P(t_1, \dots, t_n)$
- Fakten: atomare Formeln ohne Unbestimmte
- Regeln (α_i ist atomare Formel):

$$\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_m \Rightarrow \alpha_0$$

Auch leere Prämissen möglich (entsprechen **true**)!

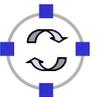


Fakten:

1. TOCHTER (Susi, Petra)
2. TOCHTER (Petra, Rita)

Regeln:

3. TOCHTER (X, Y) \wedge TOCHTER (Y, Z)
 \Rightarrow ENKELIN (X, Z)



Vereinfachte Darstellung:

- ❑ Finde Belegung der Unbestimmten einer Regel, so dass auf der linken Seite (Prämisse) bekannte Fakten stehen
- ❑ Rechte Seite ergibt dann neuen Fakt
- ❑ Logische Regel: *Modus Ponens*
(lat. ponere "stellen, setzen": setzende Schlussfigur, d.h. Schlussfigur, bei der eine positive Aussage hergeleitet wird)

$$\frac{P \Rightarrow Q; P}{Q}$$



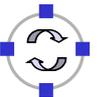
Fakten: 1. TOCHTER(Susi, Petra)
2. TOCHTER(Petra, Rita)
Regeln: 3. TOCHTER(X, Y) \wedge TOCHTER(Y, Z)
 \Rightarrow ENKELIN(X, Z)

Belegung

$X \mapsto$ Susi, $Y \mapsto$ Petra, $Z \mapsto$ Rita

ergibt Fakt

ENKELIN(Susi, Rita)



- Ein **deduktiver Algorithmus** D ist eine Menge von Fakten und Regeln.
- $F(D)$ sind alle direkt oder indirekt aus D **ableitbaren** Fakten.
- Eine **Anfrage** γ ist eine Konjunktion von atomaren Formeln.

$$\gamma = \alpha_1 \wedge \dots \wedge \alpha_m$$

- Eine Antwort ist eine Belegung der Unbestimmten in γ , bei der aus allen α_i Fakten aus $F(D)$ werden.



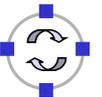
Fakten:

$$\text{SUC}(n, n + 1) \text{ für alle } n \in \mathbb{N}$$

Regeln:

$$(1) \Rightarrow \text{ADD}(X, 0, X)$$

$$(2) \text{ ADD}(X, Y, Z) \wedge \text{SUC}(Y, V) \wedge \text{SUC}(Z, W) \\ \Rightarrow \text{ADD}(X, V, W)$$



- ❑ $\text{ADD}(3, 2, 5)$? liefert **true**

Ableitung:

Regel (1) mit $X=3$;

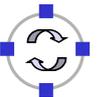
Regel (2) mit Belegung $3,0,3,1,4$; // angegeben als x, y, z, v, w

Regel (2) mit Belegung $3,1,4,2,5$;

- ❑ $\text{ADD}(3, 2, X)$? liefert $X \mapsto 5$
- ❑ $\text{ADD}(3, X, 5)$? liefert $X \mapsto 2$
- ❑ $\text{ADD}(X, Y, 5)$? liefert
 $(X, Y) \in \{(0, 5), (1, 4), (2, 3), (3, 2), (1, 4), (5, 0)\}$
- ❑ $\text{ADD}(X, Y, Z)$? liefert unendliches Ergebnis



- ❑ Vollständiger Algorithmus würde Zeitrahmen dieser Vorlesungsstunde sprengen
- ❑ Idee:
 1. Starte mit γ als zu untersuchende Anfragemenge
 2. Untersuche Belegungen, die
 - ❑ einen Teil von γ mit Fakten gleichsetzen bzw.
 - ❑ einen Fakt aus γ mit einer rechten Seite einer Regel gleichsetzen
 3. Wende passende Regeln “rückwärts” an (ersetze Konklusion durch Prämissen in zu untersuchender Anfragemenge)
 4. Entferne gefundene Fakten aus der AnfragemengeLetzte Schritte wiederholen solange bis γ leer ist



Vollständige Lösung:

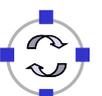
BACKTRACKING !

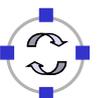
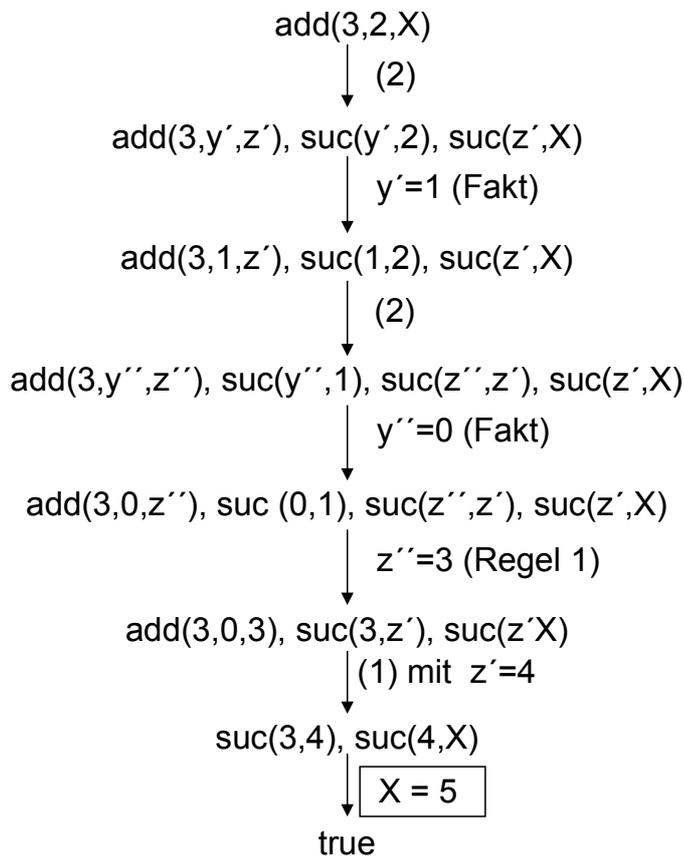
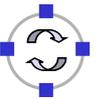
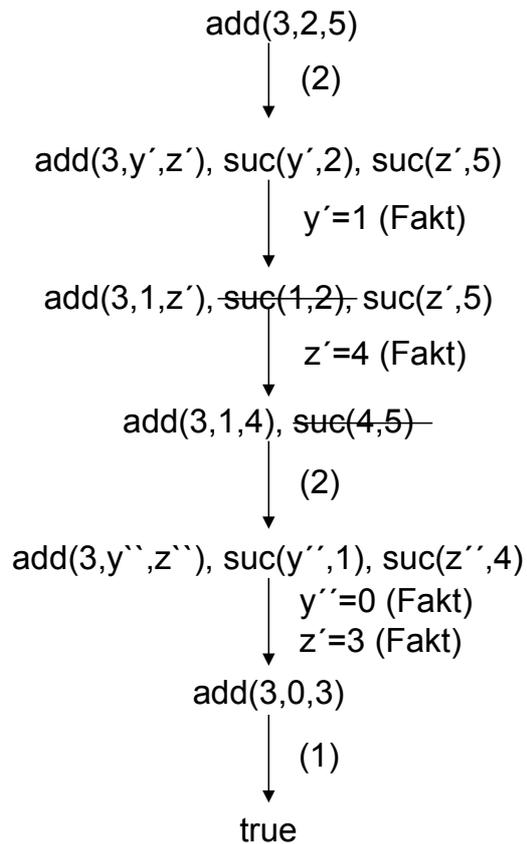
(d.h. man muss manchmal Schritte „geordnet“ zurücknehmen)

- ❑ Spezielle Form der Rekursion

Hier nur vereinfachter Formalismus:

- ❑ keine Negation;
- ❑ kein Erkennen von Endlosrekursionen,
- ❑ kein Erkennen von unendlichen Belegungsmengen etc.





- ❑ Unterschiedliche Algorithmenparadigmen
- ❑ Applikative Algorithmen:
 - ❑ Mengen von Funktionsdefinitionen
 - ❑ Rekursive Anwendung
- ❑ Imperative Algorithmen:
 - ❑ Anweisungen und Variablen
 - ❑ Syntax und Semantik
- ❑ Deduktive Algorithmen
- ❑ Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kapitel 3

