Theoretische Informatik für Wirtschaftsinformatik und Lehramt

Höhere Programmiermodelle

Priv.-Doz. Dr. Stefan Milius
stefan.milius@fau.de

Theoretische Informatik
Friedrich-Alexander Universität Erlangen-Nürnberg

SS 2016

Gliederung

Worum geht es in diesem Abschnitt? (I)

- **Letzter Abschnitt:** Definition, Arbeitsweise und Leistungsfähigkeit von Turing-Maschinen
- **Behauptung:** Turing-Maschinen sind genauso leistungsstark im Hinblick auf die Berechnung von Funktionen wie typische Programmiersprachen
- **Jetzt:** formale Untersuchung der Behauptung.
- Dazu betrachte zwei sehr stark vereinfachte Programmiersprachen:

  LOOP hat einfache Zuweisungen, Sequenzen, Fallunterscheidungen und eine Zählschleife,

  WHILE hat *zusätzlich* Schleifen mit Abbruchbedingung.

Worum geht es in diesem Abschnitt? (II)

- Es gibt Funktionen, die nicht mit LOOP programmierbar sind.

  Z.B. die *Ackermann-Funktion*.

- WHILE-Programme sind genauso „leistungsstark“, wie gängige höhere Programmiersprachen

Lernziele

- die Programmiersprachen LOOP und WHILE mit ihren jeweiligen Eigenschaften kennen und erläutern können
- arithmetische Funktionen mittels LOOP und/oder WHILE programmieren können (falls möglich)
- LOOP- bzw. WHILE-Berechenbarkeit am Beispiel der Ackermann-Funktion diskutieren können

Programmiersprachen als Berechnungsmodelle

- **Bisher:**
  Funktionen auf allgemeinen Wortmengen

  Modell der (Turing-)Berechenbarkeit:

  Berechnungen von Funktionen werden in einer relativ primitiven Maschinensprache programmiert

- **Jetzt:**
  arithmetische Funktionen

  andere Maschinenmodelle, die den Grundprinzipien realer Computer näher sind

- wichtigstes Modell:
  Registermaschine (RAM = „Random Access Machine“)

Registermaschine (RAM)

- (potentiell unendlicher) Speicher mit adressierbaren *Registern* (*Speicherzellen*):

  jedes Register speichert eine natürliche Zahl

- adressierbarer Programmspeicher:

  speichert Befehle einer Maschinensprache

- Programmierung auf einer Art "Assembler-Ebene"

  typische Befehle:
  - „Holen" einer Zahl aus dem Speicher
  - „Ablegen" einer Zahl im Speicher
  - „Sprünge" auf bestimmte Befehle
  - Befehle für einfache arithmetische Operationen, z. B.: Addition, Subtraktion, Multiplikation, (ganzzahlige) Division

RAM, TM und höhere Programmiersprachen

- TM trotz Einfachheit gleich mächtig wie Registermaschinen
- Begiff der *RAM-Berechenbarkeit* ist für eine arithmetische Funktion f formal definierbar
- Dann gilt:

$$f \text{ ist RAM-berechenbar} \iff f \text{ ist Turing-berechenbar}$$

- weitergehender Ansatz:
  - ganz von konkretem Maschinenmodell lösen
  - Funktion als (algorithmisch) berechenbar betrachten, wenn es Programm in höherer Programmiersprache zu ihrer Berechnung gibt
  - erforderlich: konkrete Modellsprache, in der essentielle algorithmische Berechnungskonzepte höherer Sprachen in möglichst einfacher Form zusammengefasst sind

LOOP-Programme

LOOP-Programme

Syntax von LOOP

Ein LOOP-**Programm** ist eine nicht-leere endliche Folge von **Anweisungen** (jeweils durch Semikolon „;“ getrennt).

Jede Anweisung ist entweder eine **Zuweisung** der Form

$$a := 0 \quad \text{oder} \quad a := a' \quad \text{oder} \quad a := a' + 1$$

oder eine **Wiederholungsanweisung (loop-Schleife)** der Form

$$\textbf{loop } a \textbf{ do } P \textbf{ enddo},$$

wobei P wieder ein LOOP-Programm ist und
a und a' (nicht notwendig verschiedene) **Variablen** aus einer (fest vorgegebenen) Variablenmenge $\{a_0, a_1, a_2, a_3, \ldots\}$ sind.

Semantik von LOOP (I)

- Ein **(Variablen-) Zustand** ist eine Belegung aller Variablen $a_0, a_1, a_2, a_3, \ldots$ mit jeweils einer natürlichen Zahl als Wert.
  (**Formal:** eine Funktion $B : \{a_0, a_1, a_2, \ldots\} \to \mathbb{N}$)
- Ein **Ablauf** eines LOOP-Programms

  $$P_1; P_2; \ldots; P_n$$

  startet in einem Zustand und verändert diesen
  (d. h. die Werte von Variablen)
  schrittweise durch Nacheinander-Ausführung der Anweisungen $P_1, P_2, \ldots, P_n$.
- Nach Ausführung von P_n ist der Ablauf **beendet**.

Semantik von LOOP (II)

Die Wirkung der möglichen Anweisungen ist wie folgt:

- Zuweisungen:

  $a := 0$ — a wird mit dem Wert 0 belegt.

  $a := a'$ — a wird mit dem (aktuellen) Wert von a' belegt.

  $a := a' + 1$ — a wird mit dem um 1 erhöhten (aktuellen) Wert von a' belegt.

  Werte der übrigen Variablen bleiben jeweils unverändert

- **loop**-Schleifen: (**loop** a **do** P **enddo**)

  P wird „Wert von a zu Beginn“-mal ausgeführt

  loop-Schleifen entsprechen den üblichen **for**-Schleifen

Semantik von LOOP (III)

Beobachtung

Da jede Schleife nur endlich viele Schritte ausführt, wird jeder Ablauf eines LOOP-Programms beendet.

Implementierung weiterer Sprachkonstrukte (I)

a, b und c seien Variablen und $n \in \mathbb{N}$:

Befehl	Implementierung
$a := n$	$a := 0; \underbrace{a := a + 1; \ldots; a := a + 1}_{n\text{-mal}}$
$a := b + c$	$a := b;$ **loop** c **do** $a := a + 1$ **enddo**
if $a = 0$ **then** P_1 **else** P_2 **endif**	$b := 1;$ **loop** a **do** $b := 0$ **enddo**; $c := 0;$ **loop** a **do** $c := 1$ **enddo**; **loop** b **do** P_1 **enddo**; **loop** c **do** P_2 **enddo**

Implementierung weiterer Sprachkonstrukte (II)

if $a = 0$ **then** P_1
else P_2 **endif**

1: $b := 1$; **loop** a **do** $b := 0$ **enddo**;
2: $c := 0$; **loop** a **do** $c := 1$ **enddo**;
3: **loop** b **do** P_1 **enddo**;
4: **loop** c **do** P_2 **enddo**

Zeile	a	b	c	Ergebnis
1	0	1	nicht def.	
2	0	1	0	
3	0	1	0	P_1 wird ausgeführt
4	0	1	0	P_2 wird nicht ausgeführt
1	1	0	nicht def.	
2	1	0	1	
3	1	0	1	P_1 wird nicht ausgeführt
4	1	0	1	P_2 wird ausgeführt

LOOP-berechenbar (I)

Definition

Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine Funktion.

Ein LOOP-Programm P **berechnet** f, wenn gilt:

Wird der Ablauf von P in einem Zustand gestartet, in dem die Variablen $a_1, \ldots, a_k$ mit $n_1, \ldots, n_k$ und alle übrigen Variablen mit 0 belegt sind, so hat die Variable a_0 in dem nach Beendigung des Ablaufs erreichten Zustand den Wert $f(n_1, \ldots, n_k)$.

f heißt LOOP-**berechenbar**, wenn es ein LOOP-Programm gibt, das f berechnet.

LOOP-berechenbar (II)

Beispiel

Funktion $f(x) = x \dot{-} 1$ wobei die
arithmetische Differenz $\dot{-}$ allgemein definiert ist durch

$$x \dot{-} y = \begin{cases} x - y, & \text{falls } x > y \\ 0 & \text{sonst.} \end{cases}$$

LOOP-Programm für f (a_0: Ergebnis; a_1: x; a_2: Hilfsvariable):

if $a_1 = 0$ **then** $a_0 := 0$
else $a_2 := 0$; **loop** a_1 **do** $a_0 := a_2$; $a_2 := a_2 + 1$ **enddo**
endif

Außer $f(x) = x \dot{-} 1$ sind auch alle sonstigen „üblichen“ arithmetischen Funktionen (Prädikate) LOOP-berechenbar.

LOOP-berechenbar (III)

Beispiel (Fortsetzung)

- Berechne $f(3) = 3 \dot{-} 1 = 2$.
- Programm: (a_0 : Ergebnis; $a_1 : x$; a_2 : Hilfsvariable):

  if $a_1 = 0$ **then** $a_0 := 0$
  else $a_2 := 0$; **loop** a_1 **do** $a_0 := a_2$; $a_2 := a_2 + 1$ **enddo**
  endif
- Weise a_1 den Wert 3 zu.
- Durchlauf der **loop**-Schleife (3-mal):
  - 1. Durchlauf: $a_0 := 0$; $a_2 := 1$
  - 2. Durchlauf: $a_0 := 1$; $a_2 := 2$
  - 3. Durchlauf: $a_0 := 2$; $a_2 := 3$

LOOP-berechenbare Funktionen und Prädikate (I)

Satz 7.1

a) Folgende Funktionen sind LOOP-berechenbar:

$$x + y, \quad x \dot{-} y, \quad x * y, \quad x \operatorname{div} y, \quad x \bmod y, \quad x^y,$$
$$\max(x, y), \quad \min(x, y), \quad |x - y|, \quad x!.$$

b) Folgende Prädikate sind LOOP-berechenbar:

$$x = y, \quad x \neq y, \quad x < y, \quad x \leq y, \quad x > y, \quad x \geq y,$$
$$x|y, \quad \text{istprim}(x).$$

Hinweise: $x|y$ bedeutet „x ist Teiler von y“

Ein Prädikat ist eine Funktion $p : \mathbb{N}^k \to \{0, 1\} \subseteq \mathbb{N}$.

LOOP-berechenbare Funktionen und Prädikate (II)

Beispiele

- Allgemeine arithmetische Differenz $f(x) = x \dot{-} y$:
  (a_0: Ergebnis; a_1: x; a_2: y)

  $a_0 := a_1$;
  loop a_2 **do** $a_0 := a_0 \dot{-} 1$ **enddo**

- Prädikat $x = y$: (a_0: Ergebnis mit $a_0 = 0$ bzw. 1 bedeutet „false“ bzw. „true“; a_1: x; a_2: y; a_h: Hilfsvariable)

  $a_0 := a_1 \dot{-} a_2$;
  $a_h := a_2 \dot{-} a_1$;
  $a_0 := a_0 + a_h$;
  if $a_0 = 0$ **then** $a_0 := 1$ **else** $a_0 := 0$ **endif**

LOOP-berechenbare Funktionen und Prädikate (III)

Satz 7.1 (Fortsetzung)

c) Sei $g : \mathbb{N}^{k+1} \to \mathbb{N}$ eine LOOP-berechenbare Funktion.

Dann gelten auch folgende Aussagen:

Funktionswertsummen der Funktion g sind LOOP-berechenbar:

$$f(x, y, x_1, \ldots, x_k) = \sum_{i=x}^{y} g(x_1, \ldots, x_k, i)$$

Funktionswertprodukte der Funktion g sind LOOP-berechenbar:

$$f(x, y, x_1, \ldots, x_k) = \prod_{i=x}^{y} g(x_1, \ldots, x_k, i),$$

LOOP-berechenbare Funktionen und Prädikate (IV)

Satz 7.1 (Fortsetzung)

c) Seien $g_1, \ldots, g_m$ LOOP-berechenbare Funktionen und $p_1, \ldots, p_{m-1}$ LOOP-berechenbare Prädikate.

Dann gilt auch folgende Aussage:

Eine fallweise aus $g_1, \ldots, g_m$ und $p_1, \ldots, p_{m-1}$ konstruierte Funktion f ist LOOP-berechenbar:

$$f(x_1, \ldots, x_k) = \begin{cases} g_1(x_1, \ldots, x_k), & \text{falls } p_1(x_1, \ldots, x_k) = 1 \\ \vdots & \\ g_{m-1}(x_1, \ldots, x_k), & \text{falls } p_{m-1}(x_1, \ldots, x_k) = 1 \\ g_m(x_1, \ldots, x_k), & \text{sonst} \end{cases}$$

LOOP-berechenbare Funktionen und Prädikate (V)

Satz 7.1 (Fortsetzung)

d) Sind q und r LOOP-berechenbare Prädikate,
so sind auch die folgenden Prädikate LOOP-berechenbar:

$$p(x_1, \ldots, x_k) = 1 \iff q(x_1, \ldots, x_k) = 0,$$
$$p(x_1, \ldots, x_k) = 1 \iff q(x_1, \ldots, x_k) = 1 \wedge r(x_1, \ldots, x_k) = 1,$$
$$p(x_1, \ldots, x_k) = 1 \iff q(x_1, \ldots, x_k) = 1 \vee r(x_1, \ldots, x_k) = 1$$

Die Ackermann-Funktion (I)

Bisher: „übliche“ arithmetische Funktionen und Prädikate sind alle LOOP-berechenbar

Frage: gibt es überhaupt (totale) Funktionen, die in „intuitivem Sinne“ berechenbar sind (etwa in einer Programmiersprache programmierbar), die jedoch *nicht* LOOP-berechenbar sind?

Antwort: JA! Zum Beispiel die **Ackermann-Funktion**:

$$\text{ack} : \mathbb{N}^2 \to \mathbb{N}$$

$$\begin{array}{lcll} \text{ack}(0, y) & = & y + 1 & \\ \text{ack}(x, 0) & = & \text{ack}(x - 1, 1) & \text{für } x > 0 \\ \text{ack}(x, y) & = & \text{ack}(x - 1, \mathit{ack}(x, y - 1)) & \text{für } x, y > 0 \end{array}$$

Die Ackermann-Funktion (II)

$\mathrm{ack}(x,y)$	$y=$	0	1	2	3	4	5	6	$\cdots$	
$x=$										
0		1	2	3	4	5	6	7	$\cdots$	$y+1$
1		2	3	4	5	6	7	8	$\cdots$	$y+2$
2		3	5	7	9	11	13	15	$\cdots$	$2y+3$
3		5	13	29	61	125	253	509	$\cdots$	$2^{y+3}-3$
4		13	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

$$\begin{aligned}
\mathrm{ack}(3,0) &= \mathrm{ack}(2,1) = \mathrm{ack}(1,\mathrm{ack}(2,0)) = \mathrm{ack}(1,3) \\
&= \mathrm{ack}(0,\mathrm{ack}(1,2)) = \mathrm{ack}(0,4) = 5 \\
\mathrm{ack}(2,0) &= \mathrm{ack}(1,1) = \mathrm{ack}(0,\mathrm{ack}(1,0)) = \mathrm{ack}(0,2) = 3 \\
\mathrm{ack}(1,0) &= \mathrm{ack}(0,1) = 2 \\
\mathrm{ack}(1,2) &= \mathrm{ack}(0,\mathrm{ack}(1,1)) = \mathrm{ack}(0,3) = 4 \\
\mathrm{ack}(1,1) &= \mathrm{ack}(0,\mathrm{ack}(1,0)) = \mathrm{ack}(0,2) = 3
\end{aligned}$$

Eigenschaften der Ackermann-Funktion (I)

$$
\begin{aligned}
\mathrm{ack}(1, y) &= y + 2 \\
\mathrm{ack}(2, y) &= 2y + 3 \\
\mathrm{ack}(3, y) &= 2^{y+3} - 3 \\
\mathrm{ack}(4, y) &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+2 \text{ viele Potenzen}} - 3
\end{aligned}
$$

Bereits

$$\mathrm{ack}(4, 2) = 2^{65536} - 3$$

ist größer als die geschätzte Anzahl der Atome im Universum.

Eigenschaften der Ackermann-Funktion (II)

Frage: Ist die Ackermann-Funktion total und intuitiv berechenbar?

Totalität (per Induktion): ack ist für alle Werte x und y definiert

- Induktionsanfang: Für $x = 0$ und beliebiges y ist $ack(x, y)$ direkt definiert.
- Induktionsschritt: Für $x > 0$ wird $\text{ack}(x, y)$ auf Kombination verschiedener Werte $\text{ack}(x', y')$ zurückgeführt, die gemäß Induktionsvoraussetzung bereits definiert sind

  (eigentlicht jetzt Induktion in y)

Intuitive Berechenbarkeit: durch Angabe eines Programms in einer höheren Programmiersprache leicht möglich (z.B. Haskell)

Also: ack ist eine totale, intuitiv berechenbare Funktion.

Eigenschaften der Ackermann-Funktion (III)

Ziel: beweisen, dass Ackermann-Funktion (mit wachsendem ersten Argument) über jede LOOP-berechenbare Funktion hinauswächst

$$\begin{array}{lcll} \mathrm{ack}(0,y) & = & y+1 & \\ \mathrm{ack}(x,0) & = & \mathrm{ack}(x-1,1) & \text{für } x>0 \\ \mathrm{ack}(x,y) & = & \mathrm{ack}(x-1,\mathit{ack}(x,y-1)) & \text{für } x,y>0 \end{array}$$

Lemma 7.2 (Monotonie)

Für alle $x, y \in \mathbb{N}$ gilt:

a) $y < \mathrm{ack}(x,y)$

b) $\mathrm{ack}(x,y) < \mathrm{ack}(x,y+1)$

c) $\mathrm{ack}(x,y+1) \leq \mathrm{ack}(x+1,y)$

d) $\mathrm{ack}(x,y) < \mathrm{ack}(x+1,y)$

Aus b) und d) folgt: $\forall x \leq x', y \leq y' : \mathrm{ack}(x,y) \leq \mathrm{ack}(x',y')$

Max. Variablenwachstum der Ackermann-Funktion (I)

- Wir ordnen jedem LOOP-Programm P eine Funktion $f_P : \mathbb{N} \to \mathbb{N}$ zu.
- Seien $a_0, a_1, \ldots, a_k$ die in P vorkommenden Variablen.
- Seien ferner n_i die jeweiligen Startwerte der Variablen a_i und n'_i deren Endwerte nach Ablauf von P.
- Wir setzen nun:

$$f_P(n) = max \left\{ \sum_{i \geq 0} n'_i \mid \sum_{i \geq 0} n_i \leq n \right\}$$

- **Informell:** f_P ist die größtmögliche Summe der Variablen-Endwerte, wenn P mit Anfangswerten gestartet wurde, deren Summe n nicht übersteigt
  (nicht verwendete Variablen haben den Wert 0 und beeinflussen Wert der Summe nicht)

Max. Variablenwachstum der Ackermann-Funktion (II)

Zeige nun, dass für k genügend groß (in Abhängigkeit von P) gilt:

$$f_P(n) < \text{ack}(k, n) \qquad \text{für alle } n \in \mathbb{N}.$$

Lemma 7.3

Für jedes LOOP-Programm P gibt es eine Konstante k, so dass für alle $n \in \mathbb{N}$ gilt:

$$f_P(n) < \text{ack}(k, n).$$

Beweis von Lemma 7.3 (I)

Induktion über den Aufbau des LOOP-Programms P

- **Fall 1:** P ist eine Wertzuweisung $a_i := 0$ oder $a_i := a_j + c$ (mit $c = 0, 1$)
  - P ist Wertzuweisung, also: zwei Variablen a_0, a_1
  - Ergebnisvariable a_0 startet mit Belegung 0
  - gemäß Definition von f_P gilt: $a_0 + a_1 \leq n$
  - Nach der Zuweisung gilt: $f_P(n) = a_0 + a_1 \leq 2n + 1$
    Schlimmstenfalls:
    $\{a_0 = 0, a_1 = n\}\ a_0 := a_1 + 1\ \{a_0 = n + 1, a_1 = n\}$
  - Wir wissen:

$$\begin{aligned} \text{ack}(1, y) &= y + 2 \\ \text{ack}(2, y) &= 2y + 3 \end{aligned}$$

  - Wähle also $k = 2$, dann gilt:

$$f_P(n) \leq 2n + 1 < 2n + 3 = \text{ack}(k, n).$$

Beweis von Lemma 7.3 (II)

- **Fall 2:** P ist eine Sequenz der Form $P_1; P_2$
  - Nach Induktionsvoraussetzung gibt es Konstanten k_1, k_2, so dass gilt:

$$f_{P_1}(n) < \text{ack}(k_1, n) \quad \text{und} \quad f_{P_2}(n) < \text{ack}(k_2, n).$$

  - Mit $k_3 := max\{k_1 - 1, k_2\}$ gilt nun die Abschätzung:

$$\begin{array}{lcll} f_P(n) & \leq & f_{P_2}(f_{P_1}(n)) & \\ & < & \text{ack}(k_2, \text{ack}(k_1, n)) & \\ & \leq & \text{ack}(k_3, \text{ack}(k_3 + 1, n)) & \text{(Monotonie, weil } k_3 + 1 \geq k_1\text{)} \\ & = & \text{ack}(k_3 + 1, n + 1) & \text{(Definition von ack)} \\ & \leq & \text{ack}(k_3 + 2, n) & \text{(Lemma 7.2.c)} \end{array}$$

  - Wähle also $k = k_3 + 2$, dann gilt: $f_P(n) < \text{ack}(k, n)$.

Beweis von Lemma 7.3 (III)

- **Fall 3:** P ist eine **loop**-Schleife der Form **loop** a_i **do** Q **enddo**.
  - Nach Induktionsvoraussetzung gibt es eine Konstante k_1, so dass $f_Q(n) < \text{ack}(k_1, n)$ für alle $n \in \mathbb{N}$ gilt.
  - O.B.d.A. komme die Variable a_i in Q nicht vor.
  - Die Funktion $f_P(n)$ wird durch eine Maximumsbildung definiert.
  - Sei $m \leq n$ eine Wahl für Variablenwert a_i, bei der das Maximum eingenommen wird.
- **Fall 3a:** $m = 0$ (d.h. Schleife wird nicht ausgeführt)
  - damit: $f_P(n) = n < \text{ack}(0, n) = n + 1$ (Def. von ack)
  - Wähle also $k = 0$, dann gilt: $f_P(n) < \text{ack}(k, n)$.
- **Fall 3b:** $m = 1$ (d.h. Schleife wird einmal ausgeführt)
  - damit: $f_P(n) \leq f_Q(n-1) + 1 \leq \text{ack}(k_1, n-1) < \text{ack}(k_1, n)$
  - Wähle also $k = k_1$, dann gilt: $f_P(n) < \text{ack}(k, n)$.

Beweis von Lemma 7.3 (IV)

- **Fall 3c:** $m \geq 2$ (d.h. Schleife wird mehrfach ausgeführt)
  - Da a_i nicht in Q vorkommt, können wir abschätzen:

$$\begin{aligned} f_P(n) &\leq \underbrace{f_Q(f_Q(\cdots f_Q}_{m\text{-mal}}(n - \underbrace{\qquad\qquad}_{\substack{m \\ a_i \text{ kommt in } Q \text{ nicht vor}}} \cdots)) + \underbrace{m}_{a_i} \\ &< \mathrm{ack}(k_1, \underbrace{f_Q(f_Q(\cdots f_Q}_{(m-1)\text{-mal}}(n-m)\cdots)) + m \\ &\vdots \\ &< \underbrace{\mathrm{ack}(k_1, \mathrm{ack}(k_1, \cdots \mathrm{ack}(k_1}_{m\text{-mal}}, n-m)\cdots)) + m \end{aligned}$$

  - Da m-mal ein Kleiner-Zeichen vorkommt (es wurde zuvor m-mal abgeschätzt) erhalten wir: ...

Beweis von Lemma 7.3 (V)

- **Fall 3c:** $m \geq 2$ (Fortsetzung)
  - Da m-mal ein Kleiner-Zeichen vorkommt (es wurde m-mal abgeschätzt), erhalten wir:

$$\begin{aligned}
f_P(n) &\leq \underbrace{\mathsf{ack}(k_1, \mathsf{ack}(k_1, \cdots \mathsf{ack}(k_1}_{m\text{-mal}}, n-m)\cdots)) \\
&< \underbrace{\mathsf{ack}(k_1, \cdots \mathsf{ack}(k_1}_{(m-1)\text{-mal}}, \mathsf{ack}(k_1+1, n-m)\cdots)) \\
&= \underbrace{\mathsf{ack}(k_1, \cdots \mathsf{ack}}_{(m-1)\text{-mal}}(k_1+1, n-m+1)\cdots)) \\
&\vdots \\
&= \mathsf{ack}(k_1+1, n-1) \quad \text{(Definition von ack)} \\
&< \mathsf{ack}(k_1+1, n) \quad \text{(Monotonie)}
\end{aligned}$$

  - Wähle also $k = k_1 + 1$, dann gilt: $f_P(n) < \mathsf{ack}(k, n)$. □

Die Ackermann-Funktion ist nicht LOOP-berechenbar

Satz 7.4

Die Ackermann-Funktion ack ist nicht LOOP-berechenbar.

Widerspruchsbeweis:

- Annahme: ack ist LOOP-berechenbar.

  Dann ist auch $g(n) = \text{ack}(n, n)$ LOOP-berechenbar.
- Sei P ein LOOP-Programm für g.
- Es gilt $g(n) \leq f_P(n)$.
- Aus Lemma 7.3 folgt: es gibt $k \in \mathbb{N}$ mit $f_P(n) < \text{ack}(k, n)$.
- Für $n = k$ gilt somit:

$$g(k) \leq f_P(k) < \text{ack}(k, k) = g(k).$$

- **Widerspruch!** Also ist die Annahme falsch, d.h. ack ist nicht LOOP-berechenbar. □

WHILE-Programme

WHILE-Programme

Feststellung: LOOP-Berechenbarkeit ist zu „schwach“, um alle intuitiv als (durch Programme) berechenbar empfundenen Funktionen zu erhalten.

Abhilfe: Erweiterung der Sprache LOOP zur Sprache WHILE:

Syntax von WHILE

Ein WHILE-**Programm** ist genauso aufgebaut wie ein LOOP-Programm.

Zusätzlich zu beliebigen LOOP-Anweisungen dürfen Wiederholungsanweisungen (**while-Schleifen**) verwendet werden:

$$\textbf{while } a \neq 0 \textbf{ do } P \textbf{ enddo}$$

(a Variable, P WHILE-Programm)

Semantik von WHILE

Ablauf eines WHILE-Programms ist definiert wie bei LOOP-Programmen.

Zusätzliche Festlegung der Wirkung einer Schleife **while** $a \neq 0$ **do** P **enddo**:

- P wird so lange nacheinander ausgeführt, wie der Wert der Variablen a nicht 0 ist.
- Wird der Wert von a nie 0, so wird die Schleife und damit der gesamte Programmablauf nicht beendet.

  (Das Programm **terminiert** für den betreffenden Startzustand **nicht**.)
- Wesentlicher Unterschied zu LOOP-Programmen: WHILE-Programme können ggf. nicht terminieren.

WHILE-berechenbar (I)

Wegen möglicher Nicht-Terminierung berechnen WHILE-Programmen i. Allg. *partielle* Funktionen:

Definition

Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine Funktion.

Ein WHILE-Programm P **berechnet** f, wenn gilt:

Wird der Ablauf von P in einem Zustand gestartet, in dem die Variablen $a_1, \ldots, a_k$ mit $n_1, \ldots, n_k$ und alle übrigen Variablen mit 0 belegt sind, so

- terminiert P genau dann, wenn $f(n_1, \ldots, n_k)$ definiert ist, und
- falls P terminiert, hat die Variable a_0 in dem nach Beendigung des Ablaufs erreichten Zustand den Wert $f(n_1, \ldots, n_k)$.

f heißt WHILE-**berechenbar**, wenn es ein WHILE-Programm P gibt, das f berechnet.

WHILE-berechenbar (II)

Jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.

WHILE-Berechenbarkeit ist mächtiger als LOOP-Berechenbarkeit, denn:

Satz 7.5

Die Ackermann-Funktion ist WHILE-berechenbar.

Beweis von Satz 7.5 (I)

Zur Erinnerung:

$$\begin{array}{lcll} \text{ack}(0, y) & = & y + 1 & \\ \text{ack}(x, 0) & = & \text{ack}(x - 1, 1) & \text{für } x > 0 \\ \text{ack}(x, y) & = & \text{ack}(x - 1, \mathit{ack}(x, y - 1)) & \text{für } x, y > 0 \end{array}$$

- Beweis des Satzen durch Angabe eines WHILE-Programms
- Für den Beweis benötigen wir einen Stack
  (d.h. folgende Unterprogramme)

```
INIT(stack)        /* initialisiert leeren Stack */
PUSH(x, stack)     /* legt x auf den Stack ab */
y := POP(stack)    /* oberstes Stackelement entfernen */
                   /* und dessen Wert in y liefern */
```

Beweis von Satz 7.5 (II)

WHILE-Programm für ack(x, y):

```
INPUT(x,y);
INIT(stack);
PUSH(x, stack);
PUSH(y, stack);
while size(stack) ≠ 1 do      ⟵ Noch ein Makro!
    y := POP(stack);
    x := POP(stack);
    if x = 0 then PUSH(y+1,stack) else
       if y = 0 then PUSH(x-1,stack); PUSH(1,stack)
       else PUSH(x-1,stack); PUSH(x,stack);
            PUSH(y-1,stack)
       endif
    endif
enddo;
result := POP(stack);
OUTPUT(result);      ⟵ Makro – schreibt Ergebnis nach a_0
```

Beweis von Satz 7.5 (III)

Frage: Wie können die Stack-Operationen durch WHILE-Programme implementiert werden?

Grundidee: Codiere den Stackinhalt mit einer geeigneten Codierungsfunktion c durch eine natürliche Zahl n

- INIT wird dann zu $n := 0$
- PUSH(a,stack) wird dann zu $n := c(a, n)$
- POP benötigt geeignete Decodierungsfunktion
- Details vgl. [Schöning, 2008, S. 108ff] □

WHILE- und Turing-Berechenbarkeit

TM sind unser „universelles“ Berechnungsmodell

Gerechtfertigt nur, wenn (zumindest) alle WHILE-berechenbaren arithmetischen Funktionen auch mit TM berechnet werden können.

Satz 7.6

Jede WHILE-berechenbare Funktion ist Turing-berechenbar.

Beweis von Satz 7.6 (Skizze) (I)

- Sei P ein WHILE-Programm, und sei r der höchste Index i der in P vorkommenden Variablen a_i.
- Ein Ablauf von P beginnt mit Werten $n_0, \ldots, n_r$ für $a_0, \ldots, a_r$ und wird gegebenenfalls mit Werten $m_0, \ldots, m_r$ beendet. (Die übrigen Variablen sind irrelevant.)
- **Zu zeigen:** solcher Ablauf kann durch eine (deterministische) TM simuliert werden, d. h. es gibt eine TM $M = (Z, \{|, \circ\}, \Gamma, \delta, z_0)$, für die gilt:
  - terminiert P nicht, so terminiert M nicht für die Eingabe $|^{n_0} \circ |^{n_1} \circ \cdots \circ |^{n_r}$
  - andernfalls gilt: $z_0|^{n_0} \circ |^{n_1} \circ \cdots \circ |^{n_r} \vdash^* z|^{m_0} \circ |^{m_1} \circ \cdots \circ |^{m_r}$ für ein $z \in Z$ mit $z|^{m_0} \circ |^{m_1} \circ \cdots \circ |^{m_r}$ final
  - (Berechnet P eine Funktion f, so lässt sich aus M leicht eine TM gewinnen, die ebenfalls f berechnet.)

Beweis von Satz 7.6 (Skizze) (II)

- Beweis durch durch Induktion über dem Aufbau von P
- O.B.d.A.: r so gewählt, dass für den höchsten Index r' von in P vorkommenden Variablen gilt: $r' \leq r$
- Also **Behauptung**:

  Ist $r \in \mathbb{N}$ und P ein WHILE-Programm, so dass $i \leq r$ gilt für alle in P vorkommenden Variablen a_i, dann gibt es eine TM M, die P simuliert.
- **Notation:** Unterstrich _ für Kopfposition

Beweis von Satz 7.6 (Skizze) (III)

1.) P sei eine Zuweisung $a_i := 0$ Arbeitsweise von M:

$\underline{n_0} \circ \cdots \circ n_i \circ \cdots \circ n_r$ **Erinnerung:** n_i kodiert durch $|^{n_i}$

$\downarrow$ zum Trennzeichen von n_i

$n_0 \circ \cdots \underline{\circ} n_i \circ \cdots \circ n_r$

$\downarrow$ $n_0 \circ \cdots \circ$ um eine Feldposition nach rechts verschieben

$n_0 \circ \cdots \underline{\circ} n_i - 1 \circ \cdots \circ n_r$

$\downarrow$ $n_0 \circ \cdots \circ$ um eine Feldposition nach rechts verschieben

$\vdots$

$\downarrow$ $n_0 \circ \cdots \circ$ um eine Feldposition nach rechts verschieben

$n_0 \circ \cdots \underline{\circ} \circ \cdots \circ n_r$

$\downarrow$ Kopf zurück

$\underline{n_0} \circ \cdots \circ \circ \cdots \circ n_r$

2.) P sei Zuweisung $a_i := a_j$ oder $a_i := a_j + 1$.
Analoge TM-Arbeitsweise wie in 1.)

Beweis von Satz 7.6 (Skizze) (IV)

3.) $P =$ **loop** a_i **do** P' **enddo**.

Induktionsvoraussetzung:

es gibt eine TM M', die P' simuliert.

O.B.d.A.: M' benutzt kein Feld auf dem Band links von n_0.

Arbeitsweise von M:

$\underline{n_0} \circ \cdots \circ n_i \circ \cdots \circ n_r$

$\downarrow$ n_i nach links kopieren

$\underline{n_i} \circ n_0 \circ \cdots \circ n_r$

$\downarrow$ n_i um 1 verringern, Kopf zu n_0

$n_i - 1 \circ \underline{n_0} \circ \cdots \circ n_r$

$\downarrow$ Rechnung gemäß M'

$n_i - 1 \circ \underline{n'_0} \circ \cdots \circ n'_r$

$\downarrow$ zur „Zählvariablen“ gehen

$\underline{n_i - 1} \circ n'_0 \circ \cdots \circ n'_r$

$\downarrow$

$\vdots$

Beweis von Satz 7.6 (Skizze) (IV)

3.) Arbeitsweise von M: (Fortsetzung)

$\vdots$

$\underline{n_i - 2} \circ n'_0 \circ \cdots \circ n'_r$

$\downarrow$

$\vdots$ ebenso

$\downarrow$

$\underline{\square} \circ m_0 \circ \cdots \circ m_r$

$\downarrow$ Kopf zu m_0, Trennzeichen vor m_0 löschen

$\underline{m_0} \circ \cdots \circ m_r$

Beweis von Satz 7.6 (Skizze) (V)

4.) $P =$ **while** $a_i \neq 0$ **do** P' **enddo**.

Induktionsvoraussetzung:

es gibt eine TM M', die P' simuliert.

Arbeitsweise von M:

$\underline{n_0} \circ \cdots \circ n_i \circ \cdots \circ n_r \qquad (*)$

$\downarrow$ zu n_i

$n_0 \circ \cdots \circ \underline{n_i} \circ \cdots \circ n_r \xrightarrow{\text{falls } n_i=0} \underline{n_0} \circ \cdots \circ n_r$

$\downarrow$ falls $n_i > 0$: zurück zu n_0

$\underline{n_0} \circ \cdots \circ n_r$

$\downarrow$ Rechnung gemäß M'

$\underline{n'_0} \circ \cdots \circ n'_r$

$\downarrow$

$\vdots$ weiter bei $(*)$

$\downarrow$

usw.

Beweis von Satz 7.6 (Skizze) (VI)

5.) $P = P_1; P_2$

Induktionsvoraussetzung:

es gibt TM M_1 und M_2, die P_1 und P_2 simulieren.

Arbeitsweise von M:

$\underline{n_0} \circ \cdots \circ n_r$

$\downarrow$

$\downarrow$ Rechnung gemäß M_1

$\underline{n'_0} \circ \cdots \circ n'_r$

$\downarrow$ Rechnung gemäß M_2 (falls M_1 terminiert)

$\underline{m_0} \circ \cdots \circ m_r$ (oder keine Terminierung von M_2)

Insgesamt leistet M das Gewünschte. □

Zusammenfassung