

Vorlesung

Theoretische Informatik II

Bernhard Beckert

Institut für Informatik



Wintersemester 2007/2008

Diese Vorlesungsmaterialien basieren zum Teil auf den Folien zu den Vorlesungen von

Katrin Erk (gehalten an der Universität Koblenz-Landau)

Jürgen Dix (gehalten an der TU Clausthal)

Christoph Kreitz (gehalten an der Universität Potsdam)

Ihnen gilt mein herzlicher Dank.

– *Bernhard Beckert, Oktober 2007*

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- LOOP-Programme
- WHILE-Programme
- GOTO-Programme
- Beziehungen zwischen LOOP, WHILE, GOTO
- Beziehungen zwischen Registermaschinen und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- **LOOP-Programme**
- WHILE-Programme
- GOTO-Programme
- Beziehungen zwischen LOOP, WHILE, GOTO
- Beziehungen zwischen Registermaschinen und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- LOOP-Programme
- **WHILE-Programme**
- GOTO-Programme
- Beziehungen zwischen LOOP, WHILE, GOTO
- Beziehungen zwischen Registermaschinen und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- LOOP-Programme
- WHILE-Programme
- **GOTO-Programme**
- Beziehungen zwischen LOOP, WHILE, GOTO
- Beziehungen zwischen Registermaschinen und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- LOOP-Programme
- WHILE-Programme
- GOTO-Programme
- **Beziehungen zwischen LOOP, WHILE, GOTO**
- Beziehungen zwischen Registermaschinen und Turingmaschinen

Inhalt von Teil II

- Registermaschinen (Random Access Machines)
- LOOP-Programme
- WHILE-Programme
- GOTO-Programme
- Beziehungen zwischen LOOP, WHILE, GOTO
- **Beziehungen zwischen Registermaschinen und Turingmaschinen**

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Teil II

Registermaschinen

- 1 Registermaschinen**
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Im Vergleich zu Turingmaschinen

- Als Berechnungsmodell gleichwertige Grundlage für Berechenbarkeitstheorie (weniger auch Komplexitätstheorie)
- Vorteil: abstrakter, einfacher zu verstehende Programme

Ähnlich wie ...

- Imperativer Kern von Programmiersprachen
- Pseudo-Code

Im Vergleich zu Turingmaschinen

- Als Berechnungsmodell gleichwertige Grundlage für Berechenbarkeitstheorie (weniger auch Komplexitätstheorie)
- **Vorteil: abstrakter, einfacher zu verstehende Programme**

Ähnlich wie ...

- Imperativer Kern von Programmiersprachen
- Pseudo-Code

Im Vergleich zu Turingmaschinen

- Als Berechnungsmodell gleichwertige Grundlage für Berechenbarkeitstheorie (weniger auch Komplexitätstheorie)
- Vorteil: abstrakter, einfacher zu verstehende Programme

Ähnlich wie ...

- Imperativer Kern von Programmiersprachen
- Pseudo-Code

Im Vergleich zu Turingmaschinen

- Als Berechnungsmodell gleichwertige Grundlage für Berechenbarkeitstheorie (weniger auch Komplexitätstheorie)
- Vorteil: abstrakter, einfacher zu verstehende Programme

Ähnlich wie ...

- Imperativer Kern von Programmiersprachen
- Pseudo-Code

Im Vergleich zu Turingmaschinen

- Als Berechnungsmodell gleichwertige Grundlage für Berechenbarkeitstheorie (weniger auch Komplexitätstheorie)
- Vorteil: abstrakter, einfacher zu verstehende Programme

Ähnlich wie ...

- Imperativer Kern von Programmiersprachen
- **Pseudo-Code**

Beispiel Pseudo-Code

Berechnung von $a \bmod b$

```
 $r := a;$   
while  $r \geq b$  do  
   $r := r - b$   
end;  
return  $r$ 
```

Fragen

- Welche Konstrukte: if? while? goto?
- Welche Datentypen: Integers? Strings?
- Welche Datenstrukturen: Objekte? Arrays?
- Welche atomaren Befehle?
- Wie funktioniert Ein- und Ausgabe?

Fragen

- Welche Konstrukte: if? while? goto?
- Welche Datentypen: Integers? Strings?
- Welche Datenstrukturen: Objekte? Arrays?
- Welche atomaren Befehle?
- Wie funktioniert Ein- und Ausgabe?

Fragen

- Welche Konstrukte: if? while? goto?
- Welche Datentypen: Integers? Strings?
- Welche Datenstrukturen: Objekte? Arrays?
- Welche atomaren Befehle?
- Wie funktioniert Ein- und Ausgabe?

Fragen

- Welche Konstrukte: if? while? goto?
- Welche Datentypen: Integers? Strings?
- Welche Datenstrukturen: Objekte? Arrays?
- Welche atomaren Befehle?
- Wie funktioniert Ein- und Ausgabe?

Fragen

- Welche Konstrukte: if? while? goto?
- Welche Datentypen: Integers? Strings?
- Welche Datenstrukturen: Objekte? Arrays?
- Welche atomaren Befehle?
- **Wie funktioniert Ein- und Ausgabe?**

Festlegungen (informell)

Konstrukte: Unterschiedliche Varianten
loop oder while oder if+goto

Datentyp: Die natürlichen Zahlen
(wesentlicher Unterschied zu realen Computern)

Datenstrukturen: Unbeschränkte aber endliche Zahl von Registern
bezeichnet mit x_1, x_2, x_3, \dots ,
enthalten jeweils eine natürliche Zahl
(keine Arrays, Objekte o. ä)

Atomare Befehle: Inkrementieren und Dekrementieren eines Registers

Eingabe/Ausgabe: n Eingabewerte in den ersten n Registern
(alle anderen Register am Anfang 0),
Ausgabe im Register $n + 1$

Festlegungen (informell)

Konstrukte: Unterschiedliche Varianten
loop oder while oder if+goto

Datentyp: Die natürlichen Zahlen
(wesentlicher Unterschied zu realen Computern)

Datenstrukturen: Unbeschränkte aber endliche Zahl von Registern
bezeichnet mit x_1, x_2, x_3, \dots ,
enthalten jeweils eine natürliche Zahl
(keine Arrays, Objekte o. ä)

Atomare Befehle: Inkrementieren und Dekrementieren eines Registers

Eingabe/Ausgabe: n Eingabewerte in den ersten n Registern
(alle anderen Register am Anfang 0),
Ausgabe im Register $n + 1$

Festlegungen (informell)

Konstrukte: Unterschiedliche Varianten
loop oder while oder if+goto

Datentyp: Die natürlichen Zahlen
(wesentlicher Unterschied zu realen Computern)

Datenstrukturen: Unbeschränkte aber endliche Zahl von Registern
bezeichnet mit x_1, x_2, x_3, \dots ,
enthalten jeweils eine natürliche Zahl
(keine Arrays, Objekte o. ä)

Atomare Befehle: Inkrementieren und Dekrementieren eines Registers

Eingabe/Ausgabe: n Eingabewerte in den ersten n Registern
(alle anderen Register am Anfang 0),
Ausgabe im Register $n + 1$

Festlegungen (informell)

Konstrukte: Unterschiedliche Varianten
loop oder while oder if+goto

Datentyp: Die natürlichen Zahlen
(wesentlicher Unterschied zu realen Computern)

Datenstrukturen: Unbeschränkte aber endliche Zahl von Registern
bezeichnet mit x_1, x_2, x_3, \dots ,
enthalten jeweils eine natürliche Zahl
(keine Arrays, Objekte o. ä)

Atomare Befehle: Inkrementieren und Dekrementieren eines Registers

Eingabe/Ausgabe: n Eingabewerte in den ersten n Registern
(alle anderen Register am Anfang 0),
Ausgabe im Register $n + 1$

Festlegungen (informell)

Konstrukte: Unterschiedliche Varianten
loop oder while oder if+goto

Datentyp: Die natürlichen Zahlen
(wesentlicher Unterschied zu realen Computern)

Datenstrukturen: Unbeschränkte aber endliche Zahl von Registern
bezeichnet mit x_1, x_2, x_3, \dots ,
enthalten jeweils eine natürliche Zahl
(keine Arrays, Objekte o. ä)

Atomare Befehle: Inkrementieren und Dekrementieren eines Registers

Eingabe/Ausgabe: n Eingabewerte in den ersten n Registern
(alle anderen Register am Anfang 0),
Ausgabe im Register $n + 1$

Definition 4.1 (LOOP-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind LOOP-Programme (und LOOP-Befehle) für alle Register x_j

Wenn P_1, P_2 LOOP-Programme sind, dann ist auch

- $P_1; P_2$

ein LOOP-Programm

Wenn P ein LOOP-Programm ist und x_j ein Register, dann ist

- $\text{loop } x_j \text{ do } P \text{ end}$

ein LOOP-Programm (und ein LOOP-Befehl)

Definition 4.1 (LOOP-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind LOOP-Programme (und LOOP-Befehle) für alle Register x_j

Wenn P_1, P_2 LOOP-Programme sind, dann ist auch

- $P_1; P_2$

ein LOOP-Programm

Wenn P ein LOOP-Programm ist und x_j ein Register, dann ist

- `loop x_j do P end`

ein LOOP-Programm (und ein LOOP-Befehl)

Definition 4.1 (LOOP-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind LOOP-Programme (und LOOP-Befehle) für alle Register x_j

Wenn P_1, P_2 LOOP-Programme sind, dann ist auch

- $P_1; P_2$

ein LOOP-Programm

Wenn P ein LOOP-Programm ist und x_j ein Register, dann ist

- `loop x_j do P end`

ein LOOP-Programm (und ein LOOP-Befehl)

Definition 4.2 (Zustand einer Registermaschine)

Ein Zustand s einer Registermaschine ist eine Abbildung

$$s : \{x_i \mid i \in \mathbb{N}\} \longrightarrow \mathbb{N}$$

die jedem Register eine natürliche Zahl als Wert zuordnet.

Registermaschinen: Definition

Definition 4.3 (Anfangszustand, Eingabe)

Seien natürliche Zahlen $m_1, \dots, m_k \in \mathbb{N}$ als Eingabe gegeben. Im Anfangszustand gilt:

- $x_i = m_i$ für $1 \leq i \leq k$
- $x_i = 0$ für $i > k$

Definition 4.4 (Ausgabe)

Terminiert eine Registermaschine, die mit Eingabe m_1, \dots, m_k gestartet wurde, in einem Zustand s_{term} , dann ist

$$s_{term}(x_{k+1})$$

die Ausgabe der Maschine.

Registermaschinen: Definition

Definition 4.3 (Anfangszustand, Eingabe)

Seien natürliche Zahlen $m_1, \dots, m_k \in \mathbb{N}$ als Eingabe gegeben. Im Anfangszustand gilt:

- $x_i = m_i$ für $1 \leq i \leq k$
- $x_i = 0$ für $i > k$

Definition 4.4 (Ausgabe)

Terminiert eine Registermaschine, die mit Eingabe m_1, \dots, m_k gestartet wurde, in einem Zustand s_{term} , dann ist

$$s_{term}(x_{k+1})$$

die Ausgabe der Maschine.

Definition 4.5

Semantik einer Registermaschine Die Semantik $\Delta(P)$ einer Registermaschine P ist eine Relation

$$\Delta(P) : S \times S$$

auf der Menge aller Zustände.

Dabei bedeutet

$$\Delta(P)(s_1, s_2)$$

dass man durch Ausführen von P vom Zustand s_1 in den Zustand s_2 gelangt.

Definition 4.5

Semantik einer Registermaschine Die Semantik $\Delta(P)$ einer Registermaschine P ist eine Relation

$$\Delta(P) : S \times S$$

auf der Menge aller Zustände.

Dabei bedeutet

$$\Delta(P)(s_1, s_2)$$

dass man durch Ausführen von P vom Zustand s_1 in den Zustand s_2 gelangt.

Definition 4.6

Eine Registermaschine P berechnet eine Funktion

$$f_P : \mathbb{N}^k \longrightarrow \mathbb{N}$$

genau dann, wenn für alle $m_1, \dots, m_k \in \mathbb{N}$ folgendes gilt:

Wenn man P im Anfangszustand für die Eingabe m_1, \dots, m_k startet, dann gilt:

- P terminiert genau dann, wenn $f(m_1, \dots, m_k)$ definiert ist
- Falls P terminiert, dann ist die Ausgabe $f(m_1, \dots, m_k)$
- (siehe nächste Folie)

Definition 4.6

Eine Registermaschine P berechnet eine Funktion

$$f_P : \mathbb{N}^k \longrightarrow \mathbb{N}$$

genau dann, wenn für alle $m_1, \dots, m_k \in \mathbb{N}$ folgendes gilt:

Wenn man P im Anfangszustand für die Eingabe m_1, \dots, m_k startet, dann gilt:

- P terminiert genau dann, wenn $f(m_1, \dots, m_k)$ definiert ist
- Falls P terminiert, dann ist die Ausgabe $f(m_1, \dots, m_k)$
- (siehe nächste Folie)

Definition 4.6

Eine Registermaschine P berechnet eine Funktion

$$f_P : \mathbb{N}^k \longrightarrow \mathbb{N}$$

genau dann, wenn für alle $m_1, \dots, m_k \in \mathbb{N}$ folgendes gilt:

Wenn man P im Anfangszustand für die Eingabe m_1, \dots, m_k startet, dann gilt:

- P terminiert genau dann, wenn $f(m_1, \dots, m_k)$ definiert ist
- Falls P terminiert, dann ist die Ausgabe $f(m_1, \dots, m_k)$
- (siehe nächste Folie)

Definition 4.6

Eine Registermaschine P berechnet eine Funktion

$$f_P : \mathbb{N}^k \longrightarrow \mathbb{N}$$

genau dann, wenn für alle $m_1, \dots, m_k \in \mathbb{N}$ folgendes gilt:

Wenn man P im Anfangszustand für die Eingabe m_1, \dots, m_k startet, dann gilt:

- P terminiert genau dann, wenn $f(m_1, \dots, m_k)$ definiert ist
- Falls P terminiert, dann ist die Ausgabe $f(m_1, \dots, m_k)$
- (siehe nächste Folie)

Registermaschinen: Berechnete Funktion

Besonderheit

Wir verlangen zusätzlich, dass eine Registermaschine, wenn sie terminiert, alle Register außer dem Ausgaberegister wieder auf ihren ursprünglichen Wert zurücksetzt (oder unverändert lässt):

- Eingaberegister x_1, \dots, x_k auf die Eingabewerte
- Register x_i für $i > k + 1$ auf 0

Folge

Eine Maschine, die auch nur für manche Eingaben diese Zusatzbedingung verletzt, berechnet gar keine Funktion.

Registermaschinen: Berechnete Funktion

Besonderheit

Wir verlangen zusätzlich, dass eine Registermaschine, wenn sie terminiert, alle Register außer dem Ausgaberegister wieder auf ihren ursprünglichen Wert zurücksetzt (oder unverändert lässt):

- Eingaberegister x_1, \dots, x_k auf die Eingabewerte
- Register x_i für $i > k + 1$ auf 0

Folge

Eine Maschine, die auch nur für manche Eingaben diese Zusatzbedingung verletzt, berechnet gar keine Funktion.

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Definition 4.7 (Berechenbare Funktion)

Eine Funktion f heißt

LOOP-berechenbar, wenn es eine Registermaschine mit einem LOOP-Programm gibt, die f berechnet

WHILE-berechenbar, wenn es eine Registermaschine mit einem WHILE-Programm gibt, die f berechnet

GOTO-berechenbar, wenn es eine Registermaschine mit einem GOTO-Programm gibt, die f berechnet

TM-berechenbar, wenn es eine Turingmaschine gibt, die f berechnet

LOOP = Menge aller LOOP-berechenbaren Funktionen

WHILE = Menge aller WHILE-berechenbaren Funktionen

GOTO = Menge aller GOTO-berechenbaren Funktionen

TM = Menge aller TM-berechenbaren Funktionen

Teil II

Registermaschinen

1 Registermaschinen

2 LOOP-Programme

3 WHILE-Programme

4 GOTO-Programme

5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme**
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Definition 5.1 (Semantik von LOOP-Programmen)

Sei P ein LOOP-Programm.

Induktive Definition von $\Delta(P)$ wie folgt:

$\Delta(x_i := x_i + 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = s_1(x_i) + 1$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(x_i := x_i - 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = \begin{cases} s_1(x_i) - 1 & \text{falls } s_1(x_i) > 0 \\ 0 & \text{falls } s_1(x_i) = 0 \end{cases}$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(P_1; P_2)(s_1, s_2)$ genau dann, wenn:

$$\Delta(P_1)(s_1, s')$$

$$\Delta(P_2)(s', s_2)$$

Definition 5.1 (Semantik von LOOP-Programmen)

Sei P ein LOOP-Programm.

Induktive Definition von $\Delta(P)$ wie folgt:

$\Delta(x_i := x_i + 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = s_1(x_i) + 1$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(x_i := x_i - 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = \begin{cases} s_1(x_i) - 1 & \text{falls } s_1(x_i) > 0 \\ 0 & \text{falls } s_1(x_i) = 0 \end{cases}$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(P_1; P_2)(s_1, s_2)$ genau dann, wenn:

$$\Delta(P_1)(s_1, s')$$

$$\Delta(P_2)(s', s_2)$$

Definition 5.2 (Semantik von LOOP-Programmen (Forts.))

$\Delta(\text{loop } x_i \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt Zustände s'_0, \dots, s'_n mit

$$- s_1(x_i) = n$$

$$- s_1 = s'_0$$

$$- s_2 = s'_n$$

$$- \Delta(P)(s'_k, s'_{k+1}) \quad \text{für } 0 \leq k < n$$

Merke

Die Anzahl der Schleifendurchläufe ist der Wert von x_i
zu **Beginn der Schleife**.

Etwaige Änderungen an x_i innerhalb der Schleifendurchläufe sind **irrelevant**

Definition 5.2 (Semantik von LOOP-Programmen (Forts.))

$\Delta(\text{loop } x_i \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt Zustände s'_0, \dots, s'_n mit

$$- s_1(x_i) = n$$

$$- s_1 = s'_0$$

$$- s_2 = s'_n$$

$$- \Delta(P)(s'_k, s'_{k+1}) \quad \text{für } 0 \leq k < n$$

Merke

Die Anzahl der Schleifendurchläufe ist der Wert von x_i
zu **Beginn der Schleife**.

Etwaige Änderungen an x_i innerhalb der Schleifendurchläufe sind **irrelevant**

Definition 5.2 (Semantik von LOOP-Programmen (Forts.))

$\Delta(\text{loop } x_i \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt Zustände s'_0, \dots, s'_n mit

$$- s_1(x_i) = n$$

$$- s_1 = s'_0$$

$$- s_2 = s'_n$$

$$- \Delta(P)(s'_k, s'_{k+1}) \quad \text{für } 0 \leq k < n$$

Merke

Die Anzahl der Schleifendurchläufe ist der Wert von x_i
zu **Beginn der Schleife**.

Etwaige Änderungen an x_i innerhalb der Schleifendurchläufe sind **irrelevant**

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatenation: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP-berechenbare Funktionen

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatination: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP-berechenbare Funktionen

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatenation: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP-berechenbare Funktionen

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatination: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP-berechenbare Funktionen

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatination: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP-berechenbare Funktionen

Theorem 5.3

Jedes LOOP-Programm terminiert für jede Eingabe.

Beweisskizze

Beweis durch Induktion über den Aufbau von Programmen, dass jedes Programm terminiert.

Induktionsanfang: Atomare Programme terminieren.

Induktionsschritt für Konkatination: einfach.

Induktionsschritt für Schleife: Da Anzahl der Durchläufe zu Anfang feststeht kann es keine Endlosschleife geben.

Korollar

Jede LOOP-berechenbare Funktion ist total.

LOOP: Definition von Zusatzbefehlen

$x_j := 0$

loop x_j do $x_j := x_j - 1$ end

$x_j := c$ für $c \in \mathbb{N}$

$x_j := 0;$
 $x_j := x_j + 1;$
 \vdots
 $x_j := x_j + 1;$ } c mal

$x_j := x_j$

loop x_j do $x_n := x_n + 1$ end;

$x_j := 0;$

loop x_n do $x_i := x_i + 1$ end

LOOP: Definition von Zusatzbefehlen

$x_j := 0$

loop x_j do $x_j := x_j - 1$ end

$x_j := c$ für $c \in \mathbb{N}$

$x_j := 0;$
 $x_j := x_j + 1;$
 \vdots
 $x_j := x_j + 1;$ } c mal

$x_j := x_j$

loop x_j do $x_n := x_n + 1$ end;

$x_j := 0;$

loop x_n do $x_i := x_i + 1$ end

LOOP: Definition von Zusatzbefehlen

$x_j := 0$

loop x_j do $x_j := x_j - 1$ end

$x_j := c$ für $c \in \mathbb{N}$

$x_j := 0;$
 $x_j := x_j + 1;$
 \vdots
 $x_j := x_j + 1;$ } c mal

$x_j := x_j$

loop x_j do $x_n := x_n + 1$ end;

$x_j := 0;$

loop x_n do $x_i := x_i + 1$ end

LOOP: Definition von Zusatzbefehlen

Im folgenden bezeichnen x_n, x_{n+1}, \dots
neue, bisher nicht verwendete Register

$x_i := e_1 \pm e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := e_1;$

$x_n := e_2;$

loop x_n do $x_i := x_i \pm 1$ end;

$x_n := 0$

$x_i := e_1 * e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := 0;$

$x_n := e_1;$

loop x_n do $x_i := x_i + e_2$ end;

$x_n := 0$

LOOP: Definition von Zusatzbefehlen

Im folgenden bezeichnen x_n, x_{n+1}, \dots
neue, bisher nicht verwendete Register

$x_i := e_1 \pm e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := e_1;$

$x_n := e_2;$

loop x_n do $x_i := x_i \pm 1$ end;

$x_n := 0$

$x_i := e_1 * e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := 0;$

$x_n := e_1;$

loop x_n do $x_i := x_i + e_2$ end;

$x_n := 0$

LOOP: Definition von Zusatzbefehlen

Im folgenden bezeichnen x_n, x_{n+1}, \dots
neue, bisher nicht verwendete Register

$x_i := e_1 \pm e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := e_1;$

$x_n := e_2;$

loop x_n do $x_i := x_i \pm 1$ end;

$x_n := 0$

$x_i := e_1 * e_2$ (e_1, e_2 arithmetische Ausdrücke)

$x_i := 0;$

$x_n := e_1;$

loop x_n do $x_i := x_i + e_2$ end;

$x_n := 0$

LOOP: Definition von Zusatzbefehlen

```
if  $x_i = 0$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := 1 - x_i$ ;
```

```
 $x_{n+1} := 1 - x_n$ ;
```

```
loop  $x_n$  do  $P_1$  end;
```

```
loop  $x_{n+1}$  do  $P_2$  end
```

```
 $x_n := 0$ ;  $x_{n+1} := 0$ 
```

```
if  $x_i \leq x_j$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := x_i - x_j$ ;
```

```
if  $x_n = 0$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := 0$ 
```

LOOP: Definition von Zusatzbefehlen

```
if  $x_i = 0$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := 1 - x_i$ ;
```

```
 $x_{n+1} := 1 - x_n$ ;
```

```
loop  $x_n$  do  $P_1$  end;
```

```
loop  $x_{n+1}$  do  $P_2$  end
```

```
 $x_n := 0$ ;  $x_{n+1} := 0$ 
```

```
if  $x_i \leq x_j$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := x_i - x_j$ ;
```

```
if  $x_n = 0$  then  $P_1$  else  $P_2$  end
```

```
 $x_n := 0$ 
```

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme**
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme**
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Definition 6.1 (WHILE-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind WHILE-Programme (und WHILE-Befehle) für alle Register x_j

Wenn P_1, P_2 WHILE-Programme sind, dann ist auch

- $P_1; P_2$

ein WHILE-Programm

Wenn P ein WHILE-Programm ist und x_j ein Register, dann ist

- `while $x_j \neq 0$ do P end`

ein WHILE-Programm (und ein WHILE-Befehl)

Definition 6.1 (WHILE-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind WHILE-Programme (und WHILE-Befehle) für alle Register x_j

Wenn P_1, P_2 WHILE-Programme sind, dann ist auch

- $P_1; P_2$

ein WHILE-Programm

Wenn P ein WHILE-Programm ist und x_j ein Register, dann ist

- `while $x_j \neq 0$ do P end`

ein WHILE-Programm (und ein WHILE-Befehl)

Definition 6.1 (WHILE-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind WHILE-Programme (und WHILE-Befehle) für alle Register x_j

Wenn P_1, P_2 WHILE-Programme sind, dann ist auch

- $P_1; P_2$

ein WHILE-Programm

Wenn P ein WHILE-Programm ist und x_j ein Register, dann ist

- `while $x_j \neq 0$ do P end`

ein WHILE-Programm (und ein WHILE-Befehl)

Definition 6.2 (Semantik von WHILE-Programmen)

Sei P ein WHILE-Programm.

Induktive Definition von $\Delta(P)$ wie folgt:

$\Delta(x_i := x_i + 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = s_1(x_i) + 1$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(x_i := x_i - 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = \begin{cases} s_1(x_i) - 1 & \text{falls } s_1(x_i) > 0 \\ 0 & \text{falls } s_1(x_i) = 0 \end{cases}$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(P_1; P_2)(s_1, s_2)$ genau dann, wenn:

$$\Delta(P_1)(s_1, s')$$

$$\Delta(P_2)(s', s_2)$$

Definition 6.2 (Semantik von WHILE-Programmen)

Sei P ein WHILE-Programm.

Induktive Definition von $\Delta(P)$ wie folgt:

$\Delta(x_i := x_i + 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = s_1(x_i) + 1$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(x_i := x_i - 1)(s_1, s_2)$ genau dann, wenn:

$$s_2(x_i) = \begin{cases} s_1(x_i) - 1 & \text{falls } s_1(x_i) > 0 \\ 0 & \text{falls } s_1(x_i) = 0 \end{cases}$$

$$s_2(x_j) = s_1(x_j) \text{ f\u00fcr } j \neq i$$

$\Delta(P_1; P_2)(s_1, s_2)$ genau dann, wenn:

$$\Delta(P_1)(s_1, s')$$

$$\Delta(P_2)(s', s_2)$$

Definition 6.3 (Semantik von WHILE-Programmen (Forts.))

$\Delta(\text{while } x_i \neq 0 \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt für ein beliebiges $n \geq 0$ Zustände s'_0, \dots, s'_n mit

– $s_1 = s'_0$

– $s_2 = s'_n$

– $\Delta(P)(s'_i, s'_{i+1})$ für $0 \leq i < n$

– $s_k(x_i) \neq 0$ für $0 \leq k < n$

– $s_n(x_i) = 0$

Merke

Die Anzahl der Schleifendurchläufe ist **nicht** zu Anfang festgelegt

Der Schleifenrumpf kann die Anzahl der Durchläufe beeinflussen

Endlosschleifen sind möglich

Definition 6.3 (Semantik von WHILE-Programmen (Forts.))

$\Delta(\text{while } x_i \neq 0 \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt für ein beliebiges $n \geq 0$ Zustände s'_0, \dots, s'_n mit

$$- s_1 = s'_0$$

$$- s_2 = s'_n$$

$$- \Delta(P)(s'_i, s'_{i+1}) \quad \text{für } 0 \leq i < n$$

$$- s_k(x_i) \neq 0 \quad \text{für } 0 \leq k < n$$

$$- s_n(x_i) = 0$$

Merke

Die Anzahl der Schleifendurchläufe ist **nicht** zu Anfang festgelegt

Der Schleifenrumpf kann die Anzahl der Durchläufe beeinflussen

Endlosschleifen sind möglich

Definition 6.3 (Semantik von WHILE-Programmen (Forts.))

$\Delta(\text{while } x_i \neq 0 \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

- es gibt für ein beliebiges $n \geq 0$ Zustände s'_0, \dots, s'_n mit
 - $s_1 = s'_0$
 - $s_2 = s'_n$
 - $\Delta(P)(s'_i, s'_{i+1})$ für $0 \leq i < n$
 - $s_k(x_i) \neq 0$ für $0 \leq k < n$
 - $s_n(x_i) = 0$

Merke

Die Anzahl der Schleifendurchläufe ist **nicht** zu Anfang festgelegt

Der Schleifenrumpf kann die Anzahl der Durchläufe beeinflussen

Endlosschleifen sind möglich

Definition 6.3 (Semantik von WHILE-Programmen (Forts.))

$\Delta(\text{while } x_i \neq 0 \text{ do } P \text{ end})(s_1, s_2)$ genau dann, wenn:

es gibt für ein beliebiges $n \geq 0$ Zustände s'_0, \dots, s'_n mit

– $s_1 = s'_0$

– $s_2 = s'_n$

– $\Delta(P)(s'_i, s'_{i+1})$ für $0 \leq i < n$

– $s_k(x_i) \neq 0$ für $0 \leq k < n$

– $s_n(x_i) = 0$

Merke

Die Anzahl der Schleifendurchläufe ist **nicht** zu Anfang festgelegt

Der Schleifenrumpf kann die Anzahl der Durchläufe beeinflussen

Endlosschleifen sind möglich

Theorem 6.4

LOOP \subseteq **WHILE** ,

d.h., jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.

Beweis

Ein LOOP-Befehl

```
loop  $x_i$  do  $P$  end
```

kann in einem WHILE-Programm simuliert werden durch

```
while  $x_i \neq 0$  do  
   $x_n := x_n + 1$ ;   $x_{n+1} := x_{n+1} + 1$ ;   $x_i := x_i - 1$   
end;  
while  $x_{n+1} \neq 0$  do  
   $x_i := x_i + 1$ ;   $x_{n+1} := x_{n+1} - 1$   
end;  
while  $x_n \neq 0$  do  
   $P$ ;   $x_n := x_n - 1$   
end;
```

Beweis

Ein LOOP-Befehl

```
loop  $x_i$  do  $P$  end
```

kann in einem WHILE-Programm simuliert werden durch

```
while  $x_i \neq 0$  do  
   $x_n := x_n + 1$ ;   $x_{n+1} := x_{n+1} + 1$ ;   $x_i := x_i - 1$   
end;  
while  $x_{n+1} \neq 0$  do  
   $x_i := x_i + 1$ ;   $x_{n+1} := x_{n+1} - 1$   
end;  
while  $x_n \neq 0$  do  
   $P$ ;   $x_n := x_n - 1$   
end;
```

Beweis

Ein LOOP-Befehl

```
loop  $x_i$  do  $P$  end
```

kann in einem WHILE-Programm simuliert werden durch

```
while  $x_i \neq 0$  do  
   $x_n := x_n + 1$ ;   $x_{n+1} := x_{n+1} + 1$ ;   $x_i := x_i - 1$   
end;  
while  $x_{n+1} \neq 0$  do  
   $x_i := x_i + 1$ ;   $x_{n+1} := x_{n+1} - 1$   
end;  
while  $x_n \neq 0$  do  
   $P$ ;   $x_n := x_n - 1$   
end;
```

Partielle WHILE-berechenbare Funktionen

Nicht-Terminierung

WHILE-Programme können Endlosschleifen enthalten. Daher:

- **WHILE-Programme terminieren nicht immer**
- WHILE-berechenbare Funktionen können für bestimmte Eingaben undefiniert sein (**partielle Funktionen sein**)

Notation

WHILE = Menge aller **totalen** WHILE-berechenbaren Funktionen

WHILE^{part} = Menge **aller** WHILE-berechenbaren Funktionen
(incl. partiellen)

Partielle WHILE-berechenbare Funktionen

Nicht-Terminierung

WHILE-Programme können Endlosschleifen enthalten. Daher:

- WHILE-Programme terminieren nicht immer
- WHILE-berechenbare Funktionen können für bestimmte Eingaben undefiniert sein (**partielle Funktionen sein**)

Notation

WHILE = Menge aller **totalen** WHILE-berechenbaren Funktionen

WHILE^{part} = Menge **aller** WHILE-berechenbaren Funktionen
(incl. partiellen)

Partielle WHILE-berechenbare Funktionen

Nicht-Terminierung

WHILE-Programme können Endlosschleifen enthalten. Daher:

- WHILE-Programme terminieren nicht immer
- WHILE-berechenbare Funktionen können für bestimmte Eingaben undefiniert sein (**partielle Funktionen sein**)

Notation

WHILE = Menge aller **totalen** WHILE-berechenbaren Funktionen

WHILE^{part} = Menge **aller** WHILE-berechenbaren Funktionen
(incl. partiellen)

Partielle WHILE-berechenbare Funktionen

Nicht-Terminierung

WHILE-Programme können Endlosschleifen enthalten. Daher:

- WHILE-Programme terminieren nicht immer
- WHILE-berechenbare Funktionen können für bestimmte Eingaben undefiniert sein (**partielle Funktionen sein**)

Notation

WHILE = Menge aller **totalen** WHILE-berechenbaren Funktionen

WHILE^{part} = Menge **aller** WHILE-berechenbaren Funktionen
(incl. partiellen)

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme**
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme**
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Definition 7.1 (Index)

Ein **Index** (Zeilennummer) ist eine Zahl $j \geq 0$.

Definition 7.2 (GOTO-Befehl, GOTO-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind GOTO-Befehle für alle Register x_j

Wenn x_j ein Register ist und j ein Index, dann ist

- $\text{if } x_j = 0 \text{ goto } j$

ein GOTO-Befehl

Wenn B_1, \dots, B_k GOTO-Befehle sind und j_1, \dots, j_k Indizes ($k \geq 1$), dann ist

- $j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$

ein GOTO-Programm

Definition 7.2 (GOTO-Befehl, GOTO-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind GOTO-Befehle für alle Register x_j

Wenn x_j ein Register ist und j ein Index, dann ist

- $\text{if } x_j = 0 \text{ goto } j$

ein GOTO-Befehl

Wenn B_1, \dots, B_k GOTO-Befehle sind und j_1, \dots, j_k Indizes ($k \geq 1$), dann ist

- $j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$

ein GOTO-Programm

Definition 7.2 (GOTO-Befehl, GOTO-Programm)

- $x_j := x_j + 1$
- $x_j := x_j - 1$

sind GOTO-Befehle für alle Register x_j

Wenn x_j ein Register ist und j ein Index, dann ist

- $\text{if } x_j = 0 \text{ goto } j$

ein GOTO-Befehl

Wenn B_1, \dots, B_k GOTO-Befehle sind und j_1, \dots, j_k Indizes ($k \geq 1$), dann ist

- $j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$

ein GOTO-Programm

Unterschied im Aufbau

- WHILE-Programme enthalten WHILE-Programme
Rekursive Definition von Syntax und Semantik
- GOTO-Programme sind eine Liste von GOTO-Befehlen
Nicht-rekursive Definition von Syntax und Semantik

Unterschied im Aufbau

- WHILE-Programme enthalten WHILE-Programme
Rekursive Definition von Syntax und Semantik
- GOTO-Programme sind eine Liste von GOTO-Befehlen
Nicht-rekursive Definition von Syntax und Semantik

Definition 7.3 (Semantik von GOTO-Programmen)

Sei

$$P = j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

ein GOTO-Programm.

Außerdem sei j_{k+1} ein Index, der nicht in P vorkommt (Programmende).

Es gilt

$$\Delta(P)(s_1, s_2)$$

genau dann, wenn es für ein beliebiges $n \geq 0$

- Zustände s'_0, \dots, s'_n
- Indizes z_0, \dots, z_n

gibt, für die folgendes gilt ...

Definition 7.3 (Semantik von GOTO-Programmen)

Sei

$$P = j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

ein GOTO-Programm.

Außerdem sei j_{k+1} ein Index, der nicht in P vorkommt (Programmende).

Es gilt

$$\Delta(P)(s_1, s_2)$$

genau dann, wenn es für ein beliebiges $n \geq 0$

- Zustände s'_0, \dots, s'_n
- Indizes z_0, \dots, z_n

gibt, für die folgendes gilt ...

Definition 7.3 (Semantik von GOTO-Programmen)

Sei

$$P = j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

ein GOTO-Programm.

Außerdem sei j_{k+1} ein Index, der nicht in P vorkommt (Programmende).

Es gilt

$$\Delta(P)(s_1, s_2)$$

genau dann, wenn es für ein beliebiges $n \geq 0$

- Zustände s'_0, \dots, s'_n
- Indizes z_0, \dots, z_n

gibt, für die folgendes gilt ...

Definition (Forts.)

- $s'_0 = s_1$
- $s'_n = s_2$
- $z_0 = j_1$
- $z_n = j_{k+1}$

Definition (Forts.)

- $s'_0 = s_1$

- $s'_n = s_2$

- $z_0 = j_1$

- $z_n = j_{k+1}$

Definition (Forts.)

- $s'_0 = s_1$
- $s'_n = s_2$
- $z_0 = j_1$
- $z_n = j_{k+1}$

Definition (Forts.)

- $s'_0 = s_1$
- $s'_n = s_2$
- $z_0 = j_1$
- $z_n = j_{k+1}$

Semantik von GOTO

Definition (Forts.)

- für $0 \leq l < n$ gilt, wobei $j_s : B_s$ die Zeile in P ist mit $j_s = z_l$:

$$\begin{aligned} \text{wenn } B = x_j := x_j + 1 : \\ s'_{l+1}(x_j) &= s'_l(x_j) + 1 \\ s'_{l+1}(x_{i'}) &= s'_l(x_{i'}) \text{ für } i' \neq j \\ z_{l+1} &= j_{s+1} \end{aligned}$$

$$\begin{aligned} \text{wenn } B = x_j := x_j - 1 : \\ s'_{l+1}(x_j) &= \begin{cases} s'_l(x_j) - 1 & \text{falls } s'_l(x_j) > 0 \\ 0 & \text{falls } s'_l(x_j) = 0 \end{cases} \\ s'_{l+1}(x_{i'}) &= s'_l(x_{i'}) \text{ für } i' \neq j \\ z_{l+1} &= j_{s+1} \end{aligned}$$

$$\begin{aligned} \text{wenn } B = \text{if } x_j = 0 \text{ goto } j_{\text{goto}} : \\ s'_{l+1} &= s'_l \\ z_{l+1} &= \begin{cases} j_{\text{goto}} & \text{falls } s_l(x_j) = 0 \\ j_{s+1} & \text{sonst} \end{cases} \end{aligned}$$

Semantik von GOTO

Definition (Forts.)

- für $0 \leq l < n$ gilt, wobei $j_s : B_s$ die Zeile in P ist mit $j_s = z_l$:

wenn $B = x_j := x_j + 1$:

$$s'_{l+1}(x_j) = s'_l(x_j) + 1$$
$$s'_{l+1}(x_{i'}) = s'_l(x_{i'}) \text{ für } i' \neq j$$
$$z_{l+1} = j_{s+1}$$

wenn $B = x_j := x_j - 1$:

$$s'_{l+1}(x_j) = \begin{cases} s'_l(x_j) - 1 & \text{falls } s'_l(x_j) > 0 \\ 0 & \text{falls } s'_l(x_j) = 0 \end{cases}$$
$$s'_{l+1}(x_{i'}) = s'_l(x_{i'}) \text{ für } i' \neq j$$
$$z_{l+1} = j_{s+1}$$

wenn $B = \text{if } x_j = 0 \text{ goto } j_{\text{goto}}$:

$$s'_{l+1} = s'_l$$
$$z_{l+1} = \begin{cases} j_{\text{goto}} & \text{falls } s'_l(x_j) = 0 \\ j_{s+1} & \text{sonst} \end{cases}$$

Definition (Forts.)

- für $0 \leq l < n$ gilt, wobei $j_s : B_s$ die Zeile in P ist mit $j_s = z_l$:

$$\begin{aligned} \text{wenn } B = x_j := x_j + 1 : \\ s'_{l+1}(x_j) &= s'_l(x_j) + 1 \\ s'_{l+1}(x_{i'}) &= s'_l(x_{i'}) \text{ für } i' \neq j \\ z_{l+1} &= j_{s+1} \end{aligned}$$

$$\begin{aligned} \text{wenn } B = x_j := x_j - 1 : \\ s'_{l+1}(x_j) &= \begin{cases} s'_l(x_j) - 1 & \text{falls } s'_l(x_j) > 0 \\ 0 & \text{falls } s'_l(x_j) = 0 \end{cases} \\ s'_{l+1}(x_{i'}) &= s'_l(x_{i'}) \text{ für } i' \neq j \\ z_{l+1} &= j_{s+1} \end{aligned}$$

$$\begin{aligned} \text{wenn } B = \text{if } x_j = 0 \text{ goto } j_{\text{goto}} : \\ s'_{l+1} &= s'_l \\ z_{l+1} &= \begin{cases} j_{\text{goto}} & \text{falls } s_l(x_j) = 0 \\ j_{s+1} & \text{sonst} \end{cases} \end{aligned}$$

Merke

Die Anzahl der Rücksprünge (\approx Schleifendurchläufe) ist **nicht** zu Anfang festgelegt

Endlosschleifen sind möglich

Notation

GOTO = Menge aller **totalen** GOTO-berechenbaren Funktionen

GOTO^{part} = Menge **aller** GOTO-berechenbaren Funktionen
(incl. partiellen)

Merke

Die Anzahl der Rücksprünge (\approx Schleifendurchläufe) ist **nicht** zu Anfang festgelegt

Endlosschleifen sind möglich

Notation

GOTO = Menge aller **totalen** GOTO-berechenbaren Funktionen

GOTO^{part} = Menge **aller** GOTO-berechenbaren Funktionen
(incl. partiellen)

Merke

Die Anzahl der Rücksprünge (\approx Schleifendurchläufe) ist **nicht** zu Anfang festgelegt

Endlosschleifen sind möglich

Notation

GOTO = Menge aller **totalen** GOTO-berechenbaren Funktionen

GOTO^{part} = Menge **aller** GOTO-berechenbaren Funktionen
(incl. partiellen)

Gleichmächtigkeit von WHILE und GOTO

Theorem 7.4

WHILE = GOTO

WHILE^{part} = GOTO^{part}

Beweis

1. **WHILE \subseteq GOTO und WHILE^{part} \subseteq GOTO^{part}:**

Es genügt zu zeigen, dass `while $x_i \neq 0$ do P` end
mit GOTO-Befehlen simuliert werden kann.

Wir können dabei annehmen, dass P kein `while` (mehr) enthält
(ersetze die `while` von innen nach außen)

Gleichmächtigkeit von WHILE und GOTO

Theorem 7.4

WHILE = GOTO

WHILE^{part} = GOTO^{part}

Beweis

1. **WHILE** \subseteq **GOTO** und **WHILE^{part}** \subseteq **GOTO^{part}**:

Es genügt zu zeigen, dass `while $x_i \neq 0$ do P` end
mit GOTO-Befehlen simuliert werden kann.

Wir können dabei annehmen, dass P kein `while` (mehr) enthält
(ersetze die `while` von innen nach außen)

Gleichmächtigkeit von WHILE und GOTO

Theorem 7.4

WHILE = GOTO

WHILE^{part} = GOTO^{part}

Beweis

1. WHILE \subseteq GOTO und WHILE^{part} \subseteq GOTO^{part}:

Es genügt zu zeigen, dass `while $x_i \neq 0$ do P` end
mit GOTO-Befehlen simuliert werden kann.

Wir können dabei annehmen, dass P kein `while` (mehr) enthält
(ersetze die `while` von innen nach außen)

Gleichmächtigkeit von WHILE und GOTO

Theorem 7.4

WHILE = GOTO

WHILE^{part} = GOTO^{part}

Beweis

1. WHILE \subseteq GOTO und WHILE^{part} \subseteq GOTO^{part}:

Es genügt zu zeigen, dass `while $x_i \neq 0$ do P` end
mit GOTO-Befehlen simuliert werden kann.

Wir können dabei annehmen, dass P kein `while` (mehr) enthält
(ersetze die `while` von innen nach außen)

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

```
while  $x_i \neq 0$  do  $P$  end
```

wird ersetzt durch

```
 $j_1$  : if  $x_i = 0$  goto  $j_3$ ;
```

```
 $P'$ ;
```

```
 $j_2$  : if  $x_n = 0$  goto  $j_1$ ; // unbedingter Sprung, da  $x_n = 0$ 
```

```
 $j_3$  :  $x_n := x_n - 1$  // NOP, nur Sprungziel
```

Dabei:

- x_n ein bisher nicht verwendetes Register
- P' entsteht aus P , indem allen Befehlen ohne Index ein beliebiger (neuer) Index vorangestellt wird

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

while $x_i \neq 0$ do P end

wird ersetzt durch

j_1 : if $x_i = 0$ goto j_3 ;

P' ;

j_2 : if $x_n = 0$ goto j_1 ; // *unbedingter Sprung, da $x_n = 0$*

j_3 : $x_n := x_n - 1$ // *NOP, nur Sprungziel*

Dabei:

- x_n ein bisher nicht verwendetes Register
- P' entsteht aus P , indem allen Befehlen ohne Index ein beliebiger (neuer) Index vorangestellt wird

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

while $x_i \neq 0$ do P end

wird ersetzt durch

j_1 : if $x_i = 0$ goto j_3 ;

P' ;

j_2 : if $x_n = 0$ goto j_1 ; // *unbedingter Sprung, da $x_n = 0$*

j_3 : $x_n := x_n - 1$ // *NOP, nur Sprungziel*

Dabei:

- x_n ein bisher nicht verwendetes Register
- P' entsteht aus P , indem allen Befehlen ohne Index ein beliebiger (neuer) Index vorangestellt wird

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

while $x_i \neq 0$ do P end

wird ersetzt durch

j_1 : if $x_i = 0$ goto j_3 ;

P' ;

j_2 : if $x_n = 0$ goto j_1 ; // *unbedingter Sprung, da $x_n = 0$*

j_3 : $x_n := x_n - 1$ // *NOP, nur Sprungziel*

Dabei:

- x_n ein bisher nicht verwendetes Register
- P' entsteht aus P , indem allen Befehlen ohne Index ein beliebiger (neuer) Index vorangestellt wird

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

2. $\text{GOTO} \subseteq \text{WHILE}$ und $\text{GOTO}^{\text{part}} \subseteq \text{WHILE}^{\text{part}}$:

Jedes GOTO-Programm

$$j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

kann durch das folgende äquivalente WHILE-Programm ersetzt werden:

```
x_index := j_1;
while x_index ≠ 0 do
  if x_index = j_1 then B'_1 end;
  if x_index = j_2 then B'_2 end;
  ⋮
  if x_index = j_k then B'_k end;
end
```

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

2. $\text{GOTO} \subseteq \text{WHILE}$ und $\text{GOTO}^{\text{part}} \subseteq \text{WHILE}^{\text{part}}$:

Jedes GOTO-Programm

$$j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

kann durch das folgende äquivalente WHILE-Programm ersetzt werden:

```
x_index := j_1;
while x_index ≠ 0 do
  if x_index = j_1 then B'_1 end;
  if x_index = j_2 then B'_2 end;
  :
  if x_index = j_k then B'_k end;
end
```

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

2. $\text{GOTO} \subseteq \text{WHILE}$ und $\text{GOTO}^{\text{part}} \subseteq \text{WHILE}^{\text{part}}$:

Jedes GOTO-Programm

$$j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

kann durch das folgende äquivalente WHILE-Programm ersetzt werden:

```
x_index := j_1;
while x_index ≠ 0 do
  if x_index = j_1 then B'_1 end;
  if x_index = j_2 then B'_2 end;
  ⋮
  if x_index = j_k then B'_k end;
end
```

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

Dabei für $1 \leq i \leq k$:

Falls $B_i = x_i := x_i \pm 1$:

$$B'_i = x_i := x_i \pm 1; x_{\text{index}} = j_{i+1}$$

Falls $B_i = \text{if } x_i = 0 \text{ goto } j_{\text{goto}}$:

$$B'_i = \text{if } x_i = 0 \text{ then } x_{\text{index}} = j_{\text{goto}} \\ \text{else } x_{\text{index}} = j_{i+1}$$

Außerdem:

$$j_{k+1} = 0$$

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

Dabei für $1 \leq i \leq k$:

Falls $B_i = x_j := x_j \pm 1$:

$$B'_i = x_j := x_j \pm 1; x_{\text{index}} = j_{i+1}$$

Falls $B_i = \text{if } x_j = 0 \text{ goto } j_{\text{goto}}$:

$$B'_i = \text{if } x_j = 0 \text{ then } x_{\text{index}} = j_{\text{goto}} \\ \text{else } x_{\text{index}} = j_{i+1}$$

Außerdem:

$$j_{k+1} = 0$$

Gleichmächtigkeit von WHILE und GOTO

Beweis (Forts.)

Dabei für $1 \leq i \leq k$:

Falls $B_i = x_j := x_j \pm 1$:

$$B'_i = x_j := x_j \pm 1; x_{\text{index}} = j_{i+1}$$

Falls $B_i = \text{if } x_j = 0 \text{ goto } j_{\text{goto}}$:

$$B'_i = \text{if } x_j = 0 \text{ then } x_{\text{index}} = j_{\text{goto}} \\ \text{else } x_{\text{index}} = j_{i+1}$$

Außerdem:

$$j_{k+1} = 0$$

Gleichmächtigkeit von WHILE und GOTO

Schlussfolgerung aus dem Beweis

Jede WHILE-berechenbare Funktion lässt sich durch ein **WHILE+IF-Programm** mit nur **einer Schleife** berechnen.

Weitere Schlussfolgerung

Spaghetti-Code (GOTO) ist nicht mächtiger als strukturierter Code (WHILE)

Denn:

Man kann mit dem einen das andere nachahmen.

Gleichmächtigkeit von WHILE und GOTO

Schlussfolgerung aus dem Beweis

Jede WHILE-berechenbare Funktion lässt sich durch ein **WHILE+IF-Programm** mit nur **einer Schleife** berechnen.

Weitere Schlussfolgerung

Spaghetti-Code (GOTO) ist nicht mächtiger als strukturierter Code (WHILE)

Denn:

Man kann mit dem einen das andere nachahmen.

Gleichmächtigkeit von WHILE und GOTO

Schlussfolgerung aus dem Beweis

Jede WHILE-berechenbare Funktion lässt sich durch ein **WHILE+IF-Programm** mit nur **einer Schleife** berechnen.

Weitere Schlussfolgerung

Spaghetti-Code (GOTO) ist nicht mächtiger als strukturierter Code (WHILE)

Denn:

Man kann mit dem einen das andere nachahmen.

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme**
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen

Teil II

Registermaschinen

- 1 Registermaschinen
- 2 LOOP-Programme
- 3 WHILE-Programme
- 4 GOTO-Programme
- 5 Beziehungen zwischen LOOP, WHILE, GOTO und Turingmaschinen**

Aus dem bisherigen wissen wir schon:

$$\text{LOOP} \subseteq \text{WHILE} = \text{GOTO} \subsetneq \text{WHILE}^{\text{part}} = \text{GOTO}^{\text{part}}$$

Noch zu zeigen:

- $\text{LOOP} \neq \text{WHILE}$
- $\text{WHILE} = \text{TM}$ und $\text{WHILE}^{\text{part}} = \text{TM}^{\text{part}}$

Aus dem bisherigen wissen wir schon:

$$\text{LOOP} \subseteq \text{WHILE} = \text{GOTO} \subsetneq \text{WHILE}^{\text{part}} = \text{GOTO}^{\text{part}}$$

Noch zu zeigen:

- $\text{LOOP} \neq \text{WHILE}$
- $\text{WHILE} = \text{TM}$ und $\text{WHILE}^{\text{part}} = \text{TM}^{\text{part}}$

Theorem 8.1

$$\mathbf{GOTO} \subseteq \mathbf{TM}$$

Beweisskizze

Es genügt zu zeigen, dass zu jedem GOTO-Programm

$$P = j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

eine äquivalente Turingmaschine konstruiert werden kann.

Theorem 8.1

$$GOTO \subseteq TM$$

Beweisskizze

Es genügt zu zeigen, dass zu jedem GOTO-Programm

$$P = j_1 : B_1; j_2 : B_2; \dots; j_k : B_k$$

eine äquivalente Turingmaschine konstruiert werden kann.

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

Sei l die Zahl der in P vorkommenden Register

Konstruiere eine TM M mit l Halbbändern
über dem Alphabet $\Sigma = \{\#, |\}$

Das i -te Band enthält immer soviele $|$, wie der Wert von x_i ist

M hat für jede Zeile $j_r : B_r$ von P einen Zustand s_r

Wenn M in s_r ist, macht sie das, was B_r entspricht:

- Register inkrementieren oder dekrementieren
- Sprungbedingung auswerten
- In entsprechenden Folgezustand gehen

All dies kann eine TM offensichtlich leisten

Beweisskizze (Forts.)

In „Theoretische Informatik I“ haben wir gezeigt:

Zu jeder TM mit mehreren Halbbändern gibt es eine äquivalente Standard-TM mit nur einem Halbband.

Also gibt es auch eine Standard-TM, die das Programm P simuliert.

Bemerkung

TM \subseteq GOTO und damit TM = GOTO = WHILE beweisen wir später

Beweisskizze (Forts.)

In „Theoretische Informatik I“ haben wir gezeigt:

Zu jeder TM mit mehreren Halbbändern gibt es eine äquivalente Standard-TM mit nur einem Halbband.

Also gibt es auch eine Standard-TM, die das Programm P simuliert.

Bemerkung

TM \subseteq GOTO und damit **TM = GOTO = WHILE** beweisen wir später

Lemma 8.2

Die Menge aller LOOP-Programme ist rekursiv aufzählbar.

Beweisskizze

Man kann eine Grammatik angeben, die die LOOP-Programme erzeugt.

Bemerkung

Das gilt genauso für WHILE-Programme, GOTO-Programme und Turingmaschinen.

Lemma 8.2

Die Menge aller LOOP-Programme ist rekursiv aufzählbar.

Beweisskizze

Man kann eine Grammatik angeben, die die LOOP-Programme erzeugt.

Bemerkung

Das gilt genauso für WHILE-Programme, GOTO-Programme und Turingmaschinen.

Lemma 8.2

Die Menge aller LOOP-Programme ist rekursiv aufzählbar.

Beweisskizze

Man kann eine Grammatik angeben, die die LOOP-Programme erzeugt.

Bemerkung

Das gilt genauso für WHILE-Programme, GOTO-Programme und Turingmaschinen.

LOOP \neq TM

Lemma 8.3

Es gibt eine Turingmaschine M_{LOOP} , die alle LOOP-Programme simuliert.

Genauer:

Gegeben sei eine Aufzählung P_1, P_2, P_3, \dots aller LOOP-Programme.

Wenn P_i auf Eingabe m , die Ausgabe o berechnet,

dann berechnet M_{LOOP} auf Eingabe $\langle i, m \rangle$ die Ausgabe o .

Beweisskizze

Der Beweis kann genauso geführt werden, wie der Beweis, dass es eine universelle TM gibt, die alle Turingmaschinen simuliert.

Bemerkung

Auch das gilt genauso für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Lemma 8.3

Es gibt eine Turingmaschine M_{LOOP} , die alle LOOP-Programme simuliert.

Genauer:

Gegeben sei eine Aufzählung P_1, P_2, P_3, \dots aller LOOP-Programme.

*Wenn P_i auf Eingabe m , die Ausgabe o berechnet,
dann berechnet M_{LOOP} auf Eingabe $\langle i, m \rangle$ die Ausgabe o .*

Beweisskizze

Der Beweis kann genauso geführt werden, wie der Beweis, dass es eine universelle TM gibt, die alle Turingmaschinen simuliert.

Bemerkung

Auch das gilt genauso für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Lemma 8.3

Es gibt eine Turingmaschine M_{LOOP} , die alle LOOP-Programme simuliert.

Genauer:

Gegeben sei eine Aufzählung P_1, P_2, P_3, \dots aller LOOP-Programme.

Wenn P_i auf Eingabe m , die Ausgabe o berechnet,
dann berechnet M_{LOOP} auf Eingabe $\langle i, m \rangle$ die Ausgabe o .

Beweisskizze

Der Beweis kann genauso geführt werden, wie der Beweis, dass es eine universelle TM gibt, die alle Turingmaschinen simuliert.

Bemerkung

Auch das gilt genauso für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Lemma 8.3

Es gibt eine Turingmaschine M_{LOOP} , die alle LOOP-Programme simuliert.

Genauer:

Gegeben sei eine Aufzählung P_1, P_2, P_3, \dots aller LOOP-Programme.

*Wenn P_i auf Eingabe m , die Ausgabe o berechnet,
dann berechnet M_{LOOP} auf Eingabe $\langle i, m \rangle$ die Ausgabe o .*

Beweisskizze

Der Beweis kann genauso geführt werden, wie der Beweis, dass es eine universelle TM gibt, die alle Turingmaschinen simuliert.

Bemerkung

Auch das gilt genauso für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Lemma 8.3

Es gibt eine Turingmaschine M_{LOOP} , die alle LOOP-Programme simuliert.

Genauer:

Gegeben sei eine Aufzählung P_1, P_2, P_3, \dots aller LOOP-Programme.

*Wenn P_i auf Eingabe m , die Ausgabe o berechnet,
dann berechnet M_{LOOP} auf Eingabe $\langle i, m \rangle$ die Ausgabe o .*

Beweisskizze

Der Beweis kann genauso geführt werden, wie der Beweis, dass es eine universelle TM gibt, die alle Turingmaschinen simuliert.

Bemerkung

Auch das gilt genauso für WHILE-Programme, GOTO-Programme und TMen.

Theorem 8.4

LOOP \neq TM

Beweis

Die Funktion $\psi : \mathbb{N} \rightarrow \mathbb{N}$ sei definiert durch:

$$\psi(i) = P_i(i) + 1 \quad (\text{Ausgabe von } P_i \text{ auf Eingabe } i \text{ plus } 1)$$

1. $\psi \in \text{TM}$

Ändere M_{LOOP} so ab, dass nach der Berechnung von $P_i(i)$ noch 1 addiert wird.

LOOP \neq TM

Theorem 8.4

LOOP \neq TM

Beweis

Die Funktion $\psi : \mathbb{N} \rightarrow \mathbb{N}$ sei definiert durch:

$$\psi(i) = P_i(i) + 1 \quad (\text{Ausgabe von } P_i \text{ auf Eingabe } i \text{ plus 1})$$

1. $\psi \in \text{TM}$

Ändere M_{LOOP} so ab, dass nach der Berechnung von $P_i(i)$ noch 1 addiert wird.

Theorem 8.4

LOOP \neq TM

Beweis

Die Funktion $\psi : \mathbb{N} \rightarrow \mathbb{N}$ sei definiert durch:

$$\psi(i) = P_i(i) + 1 \quad (\text{Ausgabe von } P_i \text{ auf Eingabe } i \text{ plus 1})$$

1. $\psi \in \text{TM}$

Ändere M_{LOOP} so ab, dass nach der Berechnung von $P_i(i)$ noch 1 addiert wird.

Theorem 8.4

LOOP \neq TM

Beweis

Die Funktion $\psi : \mathbb{N} \rightarrow \mathbb{N}$ sei definiert durch:

$$\psi(i) = P_i(i) + 1 \quad (\text{Ausgabe von } P_i \text{ auf Eingabe } i \text{ plus } 1)$$

1. $\psi \in \text{TM}$

Ändere M_{LOOP} so ab, dass nach der Berechnung von $P_i(i)$ noch 1 addiert wird.

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

Warum?

LOOP \neq TM

Beweis (Forts.)

2. $\psi \notin \text{LOOP}$

Angenommen, es gebe ein LOOP-Programm P_{i_0} , das ψ berechnet.

Aber:

Die Ausgabe von P_{i_0} auf Eingabe von i_0 ist

$$P_{i_0}(i_0) \neq P_{i_0}(i_0) + 1 = \psi(i_0)$$

Widerspruch!

Bemerkung

Dies gilt **nicht** für WHILE-Programme, GOTO-Programme und TMen.

Beweis beruht auf Totalität von ψ ,

denn sonst kann $P_{i_0}(i_0) = P_{i_0}(i_0) + 1$ undefiniert sein.

Aus dem bisherigen wissen wir:

- $\text{LOOP} \subsetneq \text{WHILE} = \text{GOTO} \subseteq \text{TM}$
- $\text{WHILE}^{\text{part}} = \text{GOTO}^{\text{part}} \subseteq \text{TM}^{\text{part}}$

Noch zu zeigen:

$\text{TM} \subseteq \text{WHILE}$ und $\text{TM}^{\text{part}} \subseteq \text{WHILE}^{\text{part}}$

Aus dem bisherigen wissen wir:

- $\text{LOOP} \subsetneq \text{WHILE} = \text{GOTO} \subseteq \text{TM}$
- $\text{WHILE}^{\text{part}} = \text{GOTO}^{\text{part}} \subseteq \text{TM}^{\text{part}}$

Noch zu zeigen:

$\text{TM} \subseteq \text{WHILE}$ und $\text{TM}^{\text{part}} \subseteq \text{WHILE}^{\text{part}}$