

# THEORETISCHE INFORMATIK UND LOGIK

## 3. Vorlesung: WHILE und LOOP

Hannes Straß

Folien: © Markus Krötzsch, <https://iccl.inf.tu-dresden.de/web/TheoLog2017>, CC BY 3.0 DE

TU Dresden, 11. April 2022

# Was bisher geschah . . .

## **Grundbegriffe, die wir verstehen und erklären können:**

DTM, NTM, Entscheider, Aufzähler, berechenbar/entscheidbar, semi-entscheidbar, unentscheidbar, Church-Turing-These

## **Das Unentscheidbare:**

- „An algorithm is a finite answer to an infinite number of questions.“  
(Stephen Kleene)
- Aber: Es gibt mehr Möglichkeiten, unendlich viele Fragen zu beantworten, als es Algorithmen geben kann. (Cantor)

## **Weitere wichtige Ergebnisse:**

- DTM und NTM haben die gleiche Ausdrucksstärke
- Zusammenhang Aufzähler  $\leftrightarrow$  Semi-Entscheidbarkeit
- Die Busy-Beaver-Funktion ist nicht berechenbar (sie wächst schneller als alle berechenbaren Funktionen).

Aufzeichnung startet . . .

# LOOP

# Von TMs zu Programmiersprachen

## Turingmaschinen als Berechnungsmodell

- **Pro:** Einfache, kurze Beschreibung (eine Folie)  
~> Beweise oft ebenfalls einfach und kurz
- **Kontra:** Umständliche Programmierung  
~> einfache Algorithmen erfordern tausende Einzelschritte

# Von TMs zu Programmiersprachen

## Turingmaschinen als Berechnungsmodell

- **Pro:** Einfache, kurze Beschreibung (eine Folie)  
~> Beweise oft ebenfalls einfach und kurz
- **Kontra:** Umständliche Programmierung  
~> einfache Algorithmen erfordern tausende Einzelschritte

## Programmiersprachen als Berechnungsmodell

- **Pro:** Einfache, bequeme Programmierung  
~> Großer Befehlssatz + Bibliotheken für Standardaufgaben
- **Kontra:** Umständliche Beschreibung  
(z.B. Beschreibung von C++ [ISO/IEC 14882] hat 776 Seiten)  
~> Eigenschaften oft unklar; Beweise sehr umständlich

# LOOP-Programme

**Idee:** Definiere eine imperative Programmiersprache, die dennoch sehr einfach ist.

## Features:

- Variablen  $x_0, x_1, x_2, \dots$  oder auch  $x, y, \text{variablenName}, \dots$   
alle vom Typ “natürliche Zahl”
- Wertezuweisungen der Form

$$x := y + 42 \quad \text{und} \quad x := y - 23$$

für beliebige natürliche Zahlen und Variablennamen

- “For-Schleifen”: **LOOP**  $x$  **DO** ... **END**

# LOOP-Programme: Syntax

Die Programmiersprache **LOOP** basiert auf einer unendlichen Menge **V** von Variablen und der Menge  $\mathbb{N}$  der natürlichen Zahlen. **LOOP-Programme** sind induktiv definiert:

- Die Ausdrücke

$x := y + n$  und  $x := y - n$  (Wertzuweisung)

sind LOOP-Programme für alle  $x, y \in \mathbf{V}$  und  $n \in \mathbb{N}$ .

- Wenn  $P_1$  und  $P_2$  LOOP-Programme sind, dann ist

$P_1; P_2$  (Hintereinanderausführung)

ein LOOP-Programm.

- Wenn  $P$  ein LOOP-Programm ist, dann ist

**LOOP x DO P END** (Schleife)

ein LOOP-Programm für jede Variable  $x \in \mathbf{V}$ .

**Vereinfachung:** Wir erlauben, „;“ in Programmen durch Zeilenumbrüche zu ersetzen.

# Beispiel

Das folgende LOOP-Programm addiert zum Wert von  $y$  genau  $x$ -mal die Zahl 2:

```
LOOP x DO  
   $y := y + 2$   
END
```

# Beispiel

Das folgende LOOP-Programm addiert zum Wert von  $y$  genau  $x$ -mal die Zahl 2:

```
LOOP x DO  
   $y := y + 2$   
END
```

Dies entspricht also der Zuweisung  $y := y + (2 * x)$ , die wir in LOOP nicht direkt schreiben können.

# LOOP-Programme: Semantik (1)

## Funktionsweise eines LOOP-Programms $P$ :

- **Eingabe:** Eine Liste von  $k$  natürlichen Zahlen  
(Anmerkung:  $k$  wird nicht durch das Programm festgelegt.)
- **Ausgabe:** Eine natürliche Zahl

$P$  berechnet also eine totale Funktion  $\mathbb{N}^k \rightarrow \mathbb{N}$ , für beliebige  $k$

# LOOP-Programme: Semantik (1)

## Funktionsweise eines LOOP-Programms $P$ :

- **Eingabe:** Eine Liste von  $k$  natürlichen Zahlen  
(Anmerkung:  $k$  wird nicht durch das Programm festgelegt.)
- **Ausgabe:** Eine natürliche Zahl

$P$  berechnet also eine totale Funktion  $\mathbb{N}^k \rightarrow \mathbb{N}$ , für beliebige  $k$

## Initialisierung für Eingabe $n_1, \dots, n_k$ :

- LOOP speichert für jede Variable eine natürliche Zahl als Wert
- Den Variablen  $x_1, \dots, x_k$  werden anfangs die Werte  $n_1, \dots, n_k$  zugewiesen
- Allen anderen Variablen wird der Anfangswert 0 zugewiesen

## LOOP-Programme: Semantik (2)

Nach der Initialisierung wird das LOOP-Programm abgearbeitet:

- $x := y + n$ :  
Der Variable  $x$  wird als neuer Wert die Summe des (alten) Wertes für  $y$  und der Zahl  $n$  zugewiesen.
- $x := y - n$ :  
Der Variable  $x$  wird als neuer Wert die Differenz des (alten) Wertes für  $y$  und der Zahl  $n$  zugewiesen, falls diese größer als 0 ist;  
ansonsten wird  $x$  der Wert 0 zugewiesen.
- $P_1 ; P_2$ :  
Erst wird  $P_1$  abgearbeitet, dann  $P_2$ .
- **LOOP  $x$  DO  $P$  END:**  
 $P$  wird genau  $n$ -mal ausgeführt, für den Zahlenwert  $n$ , der  $x$  anfangs zugewiesen ist.  
(Die Anzahl der Schleifendurchläufe bleibt also gleich, wenn  $P$  den Wert von  $x$  ändert.)

# LOOP-Programme: Semantik (3)

## **Ausgabe eines LOOP-Programms:**

- Das Ergebnis der Abarbeitung ist der Wert der Variable  $x_0$  nach dem Beenden der Berechnung.

# LOOP-Programme: Semantik (3)

## Ausgabe eines LOOP-Programms:

- Das Ergebnis der Abarbeitung ist der Wert der Variable  $x_0$  nach dem Beenden der Berechnung.

**Satz:** LOOP-Programme terminieren immer nach endlich vielen Schritten.

# LOOP-Programme: Semantik (3)

## Ausgabe eines LOOP-Programms:

- Das Ergebnis der Abarbeitung ist der Wert der Variable  $x_0$  nach dem Beenden der Berechnung.

**Satz:** LOOP-Programme terminieren immer nach endlich vielen Schritten.

**Beweis:** Gemäß der Definition von LOOP-Programmen per struktureller Induktion.

Induktionsanfang: Die Behauptung gilt sicherlich für Wertzuweisungen.

Weitere Fälle (Induktionsschritte):

- $P_1 ; P_2$ :  
Wenn  $P_1$  und  $P_2$  nach endlich vielen Schritten terminieren, dann auch  $P_1 ; P_2$ .
- **LOOP x DO P END:**  
Für jede mögliche Zuweisung von  $x$  wird  $P$  endlich oft wiederholt; wenn  $P$  in endlich vielen Schritten terminiert, dann also auch die Schleife. □

# Quiz: LOOP-Programm

## Funktionsweise eines LOOP-Programms $P$ :

- **Eingabe:** Eine Liste von  $k$  natürlichen Zahlen ( $k$  wird nicht durch das Programm festgelegt.)
- **Ausgabe:** Eine natürliche Zahl (Der Wert von  $x_0$  nach Abarbeitung des Programms.)

$P$  berechnet also für jedes  $k \in \mathbb{N}$  eine totale Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ .

**Quiz:** Wir betrachten folgendes LOOP-Programm: ...

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable:** “ $x := y$ ”:

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable:** “ $x := y$ ”:

```
x := y + 0
```

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable:** “ $x := y$ ”:

```
x := y + 0
```

**Wertzuweisung mit 0:** “ $x := 0$ ”:

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable:** “ $x := y$ ”:

```
x := y + 0
```

**Wertzuweisung mit 0:** “ $x := 0$ ”:

```
LOOP x DO
```

```
  x := x - 1
```

```
END
```

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable: “ $x := y$ ”:**

```
x := y + 0
```

**Wertzuweisung mit 0: “ $x := 0$ ”:**

```
LOOP x DO  
  x := x - 1  
END
```

**Wertzuweisung mit einer beliebigen konstanten Zahl: “ $x := n$ ”:**

# Programmieren in LOOP (1)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung mit Variable:** “ $x := y$ ”:

```
x := y + 0
```

**Wertzuweisung mit 0:** “ $x := 0$ ”:

```
LOOP x DO
```

```
  x := x - 1
```

```
END
```

**Wertzuweisung mit einer beliebigen konstanten Zahl:** “ $x := n$ ”:

```
x := 0
```

```
x := x + n
```

## Programmieren in LOOP (2)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung:** “ $x := y + z$ ”:

## Programmieren in LOOP (2)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung:** “ $x := y + z$ ”:

```
x := y
LOOP z DO
  x := x + 1
END
```

## Programmieren in LOOP (2)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung:** “ $x := y + z$ ”:

```
x := y
LOOP z DO
  x := x + 1
END
```

**Fallunterscheidung:** “**IF**  $x \neq 0$  **THEN**  $P$  **END**”:

## Programmieren in LOOP (2)

LOOP hat nur wenige Ausdrucksmittel, aber man kann sich leicht weitere als Makros definieren.

**Wertzuweisung:** “ $x := y + z$ ”:

```
x := y
LOOP z DO
  x := x + 1
END
```

**Fallunterscheidung:** “IF  $x \neq 0$  THEN  $P$  END”:

```
LOOP x DO y := 1 END
LOOP y DO P END
```

Dabei ist  $y$  eine frische Variable, die bisher nirgends sonst verwendet wird.

# LOOP-Berechenbare Funktionen

Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt genau dann **LOOP-berechenbar**, wenn es ein LOOP-Programm gibt, das die Funktion berechnet.

# LOOP-Berechenbare Funktionen

Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt genau dann **LOOP-berechenbar**, wenn es ein LOOP-Programm gibt, das die Funktion berechnet.

**Beispiel:** Die folgenden Funktionen sind LOOP-berechenbar:

- Addition:  $\langle x, y \rangle \mapsto x + y$  (gerade gezeigt)
- Multiplikation:  $\langle x, y \rangle \mapsto x \cdot y$  (siehe Übung)
- Potenz:  $\langle x, y \rangle \mapsto x^y$  (entsteht aus  $\cdot$  wie  $\cdot$  aus  $+$ )
- und viele andere ... (max, min, div, mod, usw.)

# LOOP jenseits von $\mathbb{N}$

# LOOP jenseits von $\mathbb{N}$

LOOP kann auch das  $x$ -te Bit der Binärkodierung von  $y$  berechnen. Dadurch kann man in LOOP (auf umständliche Weise) auch Daten verarbeiten, die keine Zahlen sind:

- (1) Kodiere beliebigen Input binär
- (2) Evaluiere die Binärkodierung als natürliche Zahl und verwende diese als Eingabe
- (3) Dekodiere den Input im LOOP-Programm

In diesem Sinne sind viele weitere Funktionen LOOP-berechenbar.

# LOOP jenseits von $\mathbb{N}$

LOOP kann auch das  $x$ -te Bit der Binärkodierung von  $y$  berechnen. Dadurch kann man in LOOP (auf umständliche Weise) auch Daten verarbeiten, die keine Zahlen sind:

- (1) Kodiere beliebigen Input binär
- (2) Evaluiere die Binärkodierung als natürliche Zahl und verwende diese als Eingabe
- (3) Dekodiere den Input im LOOP-Programm

In diesem Sinne sind viele weitere Funktionen LOOP-berechenbar.

Beispiele für LOOP-berechenbare Funktionen:

- das Wortproblem regulärer, kontextfreier und kontextsensitiver Sprachen
- alle Probleme in NP, z.B. Erfüllbarkeit aussagenlogischer Formeln
- praktisch alle „gängigen“ Algorithmen (Sortieren, Suchen, Optimieren, ...)

# Die Grenzen von LOOP

**Satz:** Es gibt berechenbare Funktionen, die nicht LOOP-berechenbar sind.

# Die Grenzen von LOOP

**Satz:** Es gibt berechenbare Funktionen, die nicht LOOP-berechenbar sind.

Das ist weniger überraschend, als es vielleicht klingt:

## **Beweis:**

- Ein LOOP-Programm terminiert immer.
- Daher ist jede LOOP-berechenbare Funktion total.
- Es gibt aber auch nicht-totale Funktionen, die berechenbar sind.  
(Z.B. die “partiellste” Funktion, die nirgends definiert ist.)

□

# LOOP-berechenbar $\neq$ berechenbar

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

# LOOP-berechenbar $\neq$ berechenbar

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

Das ist überraschend. Hilbert glaubte 1926 noch, dass alle Funktionen so berechnet werden können – quasi ein erster Versuch der Definition von Berechenbarkeit.

Hilbert definierte LOOP-Berechenbarkeit etwas anders, mit Hilfe sogenannter **primitiv rekursiver Funktionen**.

# LOOP-berechenbar $\neq$ berechenbar

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

Das ist überraschend. Hilbert glaubte 1926 noch, dass alle Funktionen so berechnet werden können – quasi ein erster Versuch der Definition von Berechenbarkeit.

Hilbert definierte LOOP-Berechenbarkeit etwas anders, mit Hilfe sogenannter **primitiv rekursiver Funktionen**.

Bewiesen wurde der Satz zuerst von zwei Studenten Hilberts:

- Gabriel Sudan (1927)
- Wilhelm Ackermann (1928)

# LOOP-berechenbar $\neq$ berechenbar

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

Das ist überraschend. Hilbert glaubte 1926 noch, dass alle Funktionen so berechnet werden können – quasi ein erster Versuch der Definition von Berechenbarkeit.

Hilbert definierte LOOP-Berechenbarkeit etwas anders, mit Hilfe sogenannter **primitiv rekursiver Funktionen**.

Bewiesen wurde der Satz zuerst von zwei Studenten Hilberts:

- Gabriel Sudan (1927)
- Wilhelm Ackermann (1928)

Jeder der beiden gab eine Funktion an (Sudan-Funktion und Ackermann-Funktion), die nicht LOOP-berechenbar ist.

Unser Beweis verwendet eine etwas andere Idee ...

# Fleißige Biber für LOOP

Die **Länge** eines LOOP-Programms ist die Anzahl an Zeichen, aus denen es besteht.

Dazu nehmen wir an:

- Zahlen werden in ihrer Dezimalkodierung geschrieben
- Variablen sind mit lateinischen Buchstaben und Ziffern benannt (wir sehen  $x_{123}$  als Schreibweise für  $x123$  an)
- Wir betrachten ; als ein Zeichen (Zeilenumbrüche werden dagegen nicht gezählt)

# Fleißige Biber für LOOP

Die **Länge** eines LOOP-Programms ist die Anzahl an Zeichen, aus denen es besteht.

Dazu nehmen wir an:

- Zahlen werden in ihrer Dezimalkodierung geschrieben
- Variablen sind mit lateinischen Buchstaben und Ziffern benannt (wir sehen  $x_{123}$  als Schreibweise für  $x123$  an)
- Wir betrachten ; als ein Zeichen (Zeilenumbrüche werden dagegen nicht gezählt)

Die Funktion  $\Sigma_{\text{LOOP}} : \mathbb{N} \rightarrow \mathbb{N}$  liefert für jede Zahl  $\ell$  die größte Zahl  $\Sigma_{\text{LOOP}}(\ell)$ , die von einem LOOP-Programm der Länge  $\leq \ell$  für eine leere Eingabe (alle Variablen sind 0) ausgegeben wird. Dabei sei  $\Sigma_{\text{LOOP}}(\ell) = 0$  falls es kein Programm der Länge  $\leq \ell$  gibt.

# Fleißige Biber für LOOP

Die **Länge** eines LOOP-Programms ist die Anzahl an Zeichen, aus denen es besteht.

Dazu nehmen wir an:

- Zahlen werden in ihrer Dezimalkodierung geschrieben
- Variablen sind mit lateinischen Buchstaben und Ziffern benannt (wir sehen  $x_{123}$  als Schreibweise für  $x123$  an)
- Wir betrachten ; als ein Zeichen (Zeilenumbrüche werden dagegen nicht gezählt)

Die Funktion  $\Sigma_{\text{LOOP}} : \mathbb{N} \rightarrow \mathbb{N}$  liefert für jede Zahl  $\ell$  die größte Zahl  $\Sigma_{\text{LOOP}}(\ell)$ , die von einem LOOP-Programm der Länge  $\leq \ell$  für eine leere Eingabe (alle Variablen sind 0) ausgegeben wird. Dabei sei  $\Sigma_{\text{LOOP}}(\ell) = 0$  falls es kein Programm der Länge  $\leq \ell$  gibt.

**Beobachtung:**  $\Sigma_{\text{LOOP}}$  ist wohldefiniert:

- Die Zahl der LOOP-Programme mit maximaler Länge  $\ell$  ist endlich.
- Unter diesen Programmen gibt es eine maximale Ausgabe.

# Beispiele

**Beispiel:** Die LOOP-Anweisung  $x_0 := y + 9$  ist das „fleißigste“ Programm für  $\ell = 7$ , d.h. es bezeugt  $\Sigma_{\text{LOOP}}(7) = 9$ .

# Beispiele

**Beispiel:** Die LOOP-Anweisung  $x_0 := y + 9$  ist das „fleißigste“ Programm für  $\ell = 7$ , d.h. es bezeugt  $\Sigma_{\text{LOOP}}(7) = 9$ .

Für  $\ell = 8$  gilt dementsprechend bereits  $\Sigma_{\text{LOOP}}(8) = 99$ .

Für  $\ell < 7$  gibt es keine Zuweisung, die  $x_0$  ändert, d.h.,  $\Sigma_{\text{LOOP}}(\ell) = 0$ .

# Beispiele

**Beispiel:** Die LOOP-Anweisung  $x_0 := y + 9$  ist das „fleißigste“ Programm für  $\ell = 7$ , d.h. es bezeugt  $\Sigma_{\text{LOOP}}(7) = 9$ .

Für  $\ell = 8$  gilt dementsprechend bereits  $\Sigma_{\text{LOOP}}(8) = 99$ .

Für  $\ell < 7$  gibt es keine Zuweisung, die  $x_0$  ändert, d.h.,  $\Sigma_{\text{LOOP}}(\ell) = 0$ .

**Bonusaufgabe:** Gibt es eine Zahl  $\ell$ , bei der  $\Sigma_{\text{LOOP}}(\ell)$  durch ein Programm berechnet wird, welches die Zahl  $\Sigma_{\text{LOOP}}(\ell)$  nicht als Konstante im Quelltext enthält? Wie könnte das entsprechende Programm aussehen?

## Beweis (1)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

# Beweis (1)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

# Beweis (1)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (1) ist leicht zu zeigen:

- Es gibt endlich viele LOOP-Programme der Länge  $\leq \ell$ .
- Man kann alle davon durchlaufen und auf einem Computer simulieren.
- Die Simulation liefert immer nach endlich vielen Schritten ein Ergebnis.
- Das Maximum aller Ergebnisse ist der Wert von  $\Sigma_{\text{LOOP}}(\ell)$ .

(Anmerkung: Wir verwenden hier einen intuitiven Berechnungsbegriff und Church-Turing.)

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)
- Sei  $P_m$  das Programm  $x_1 := x_1 + m$  (Länge:  $7 + \lceil \log_{10} m \rceil$ )

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)
- Sei  $P_m$  das Programm  $x_1 := x_1 + m$  (Länge:  $7 + \lceil \log_{10} m \rceil$ )
- Sei  $P_{++}$  das Programm  $x_0 := x_0 + 1$  (Länge: 8)

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)
- Sei  $P_m$  das Programm  $x_1 := x_1 + m$  (Länge:  $7 + \lceil \log_{10} m \rceil$ )
- Sei  $P_{++}$  das Programm  $x_0 := x_0 + 1$  (Länge: 8)
- Wir definieren  $P = P_m ; P_{\Sigma} ; P_{++}$ .

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)
- Sei  $P_m$  das Programm  $x_1 := x_1 + m$  (Länge:  $7 + \lceil \log_{10} m \rceil$ )
- Sei  $P_{++}$  das Programm  $x_0 := x_0 + 1$  (Länge: 8)
- Wir definieren  $P = P_m ; P_{\Sigma} ; P_{++}$ .  
Die Länge von  $P$  ist  $\ell = k + 17 + \lceil \log_{10} m \rceil$ , damit gilt  $\ell \leq m$ .

## Beweis (2)

**Satz:** Es gibt berechenbare totale Funktionen, die nicht LOOP-berechenbar sind.

**Beweis:** Wir zeigen zwei Teilaussagen:

- (1)  $\Sigma_{\text{LOOP}}$  ist berechenbar.
- (2)  $\Sigma_{\text{LOOP}}$  ist nicht LOOP-berechenbar.

Behauptung (2) zeigen wir per Widerspruch:

- Angenommen  $\Sigma_{\text{LOOP}}$  ist LOOP-berechenbar durch Programm  $P_{\Sigma}$ .  
Sei  $k$  die Länge von  $P_{\Sigma}$ .
- Wir wählen eine Zahl  $m$  mit  $m \geq k + 17 + \log_{10} m$  (immer möglich)
- Sei  $P_m$  das Programm  $x_1 := x_1 + m$  (Länge:  $7 + \lceil \log_{10} m \rceil$ )
- Sei  $P_{++}$  das Programm  $x_0 := x_0 + 1$  (Länge: 8)
- Wir definieren  $P = P_m ; P_{\Sigma} ; P_{++}$ .  
Die Länge von  $P$  ist  $\ell = k + 17 + \lceil \log_{10} m \rceil$ , damit gilt  $\ell \leq m$ .  
Aber  $P$  gibt die Zahl  $\Sigma_{\text{LOOP}}(m) + 1$  aus. Widerspruch. □

# WHILE

# Was fehlt?

Frage: Wieso ist LOOP zu schwach?

# Was fehlt?

Frage: Wieso ist LOOP zu schwach?

Intuitive Antwort: LOOP-Programme terminieren immer (zu vorhersehbar).

~> Wir brauchen ein weniger vorhersehbares Programmkonstrukt.

# WHILE-Programme: Syntax und Semantik

Die Programmiersprache **WHILE** basiert wie **LOOP** auf Variablen **V** und natürlichen Zahlen **N**.

**WHILE-Programme** sind induktiv definiert:

- Jedes **LOOP**-Programm ist ein **WHILE**-Programm;
- wenn  $P$  ein **WHILE**-Programm ist, dann ist

**WHILE**  $x \neq 0$  **DO**  $P$  **END**

ein **WHILE**-Programm, für jede Variable  $x \in \mathbf{V}$ .

# WHILE-Programme: Syntax und Semantik

Die Programmiersprache **WHILE** basiert wie **LOOP** auf Variablen **V** und natürlichen Zahlen **N**.

**WHILE-Programme** sind induktiv definiert:

- Jedes **LOOP**-Programm ist ein **WHILE**-Programm;
- wenn  $P$  ein **WHILE**-Programm ist, dann ist

**WHILE**  $x \neq 0$  **DO**  $P$  **END**

ein **WHILE**-Programm, für jede Variable  $x \in \mathbf{V}$ .

**Semantik** von **WHILE**  $x \neq 0$  **DO**  $P$  **END**:

$P$  wird ausgeführt, solange der aktuelle Wert von  $x$  ungleich 0 ist, wobei der Wert von  $x$  jeweils vor jeder Ausführung von  $P$  geprüft wird.

(Die Ausführung hängt also davon ab, wie  $P$  den Wert von  $x$  ändert.)

Ansonsten werden **WHILE**-Programme wie **LOOP**-Programme ausgewertet.

# WHILE: Beobachtungen

Es ist möglich, dass ein WHILE-Programm nicht terminiert, z.B.

```
x := 1
WHILE x != 0 DO
  y := y + 2
END
```

# WHILE: Beobachtungen

Es ist möglich, dass ein WHILE-Programm nicht terminiert, z.B.

```
x := 1
WHILE x != 0 DO
  y := y + 2
END
```

Wir können **LOOP x DO P END** (für ein „frisches“ z) ersetzen durch:

```
z := x
WHILE z != 0 DO
  P
  z := z - 1
END
```

Also sind LOOP-Schleifen eigentlich nicht mehr nötig.

# Quiz: WHILE-Programm

Semantik von **WHILE**  $x \neq 0$  **DO**  $P$  **END**:

$P$  wird ausgeführt, solange der aktuelle Wert von  $x$  ungleich 0 ist, wobei der Wert von  $x$  jeweils vor jeder Ausführung von  $P$  geprüft wird.

(Die Ausführung hängt also davon ab, wie  $P$  den Wert von  $x$  ändert.)

**Quiz:** Wir betrachten folgendes WHILE-Programm: ...

# WHILE-Berechenbare Funktionen

Eine partielle Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt genau dann **WHILE-berechenbar**, wenn es ein WHILE-Programm  $P$  gibt, so dass gilt:

- Falls  $f(n_1, \dots, n_k)$  definiert ist, dann terminiert  $P$  bei Eingabe  $n_1, \dots, n_k$  mit der Ausgabe  $f(n_1, \dots, n_k)$ ;
- falls  $f(n_1, \dots, n_k)$  nicht definiert ist, dann terminiert  $P$  bei Eingabe  $n_1, \dots, n_k$  nicht.

# WHILE-Berechenbare Funktionen

Eine partielle Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt genau dann **WHILE-berechenbar**, wenn es ein WHILE-Programm  $P$  gibt, so dass gilt:

- Falls  $f(n_1, \dots, n_k)$  definiert ist, dann terminiert  $P$  bei Eingabe  $n_1, \dots, n_k$  mit der Ausgabe  $f(n_1, \dots, n_k)$ ;
- falls  $f(n_1, \dots, n_k)$  nicht definiert ist, dann terminiert  $P$  bei Eingabe  $n_1, \dots, n_k$  nicht.

Das wichtigste Ergebnis zu WHILE ist nun das folgende:

**Satz:** Eine partielle Funktion ist genau dann WHILE-berechenbar, wenn sie Turing-berechenbar ist.

WHILE  $\rightarrow$  TM

**Behauptung 1:** DTMs können WHILE-Programme simulieren:

# WHILE $\rightarrow$ TM

**Behauptung 1:** DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.

# WHILE $\rightarrow$ TM

**Behauptung 1:** DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.
- Natürliche Zahlen werden auf den Bändern binär kodiert.

# WHILE $\rightarrow$ TM

## **Behauptung 1:** DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.
- Natürliche Zahlen werden auf den Bändern binär kodiert.
- DTMs können leicht (a) ein Band auf ein anderes kopieren, (b) die Zahl auf einem Band um eins erhöhen.  
 $\leadsto$  Daraus kann man schon DTMs für  $x := y + n$  erzeugen.

# WHILE $\rightarrow$ TM

## Behauptung 1: DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.
- Natürliche Zahlen werden auf den Bändern binär kodiert.
- DTMs können leicht (a) ein Band auf ein anderes kopieren, (b) die Zahl auf einem Band um eins erhöhen.  
 $\leadsto$  Daraus kann man schon DTMs für  $x := y + n$  erzeugen.
- Die Simulation von  $x := y - n$  ist analog möglich (mit zusätzlichem Test auf Gleichheit mit 0 beim Dekrementieren).

# WHILE $\rightarrow$ TM

## Behauptung 1: DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.
- Natürliche Zahlen werden auf den Bändern binär kodiert.
- DTMs können leicht (a) ein Band auf ein anderes kopieren, (b) die Zahl auf einem Band um eins erhöhen.  
 $\leadsto$  Daraus kann man schon DTMs für  $x := y + n$  erzeugen.
- Die Simulation von  $x := y - n$  ist analog möglich (mit zusätzlichem Test auf Gleichheit mit 0 beim Dekrementieren).
- Sequentielle Programmausführung  $P_1 ; P_2$  wird direkt im Zustandsgraphen der DTM umgesetzt („Hintereinanderhängen“ von TMs).

# WHILE $\rightarrow$ TM

## Behauptung 1: DTMs können WHILE-Programme simulieren:

- Wir verwenden eine Mehrband-TM, in der es für jede Variable im simulierten Programm ein eigenes Band gibt.
- Natürliche Zahlen werden auf den Bändern binär kodiert.
- DTMs können leicht (a) ein Band auf ein anderes kopieren, (b) die Zahl auf einem Band um eins erhöhen.  
 $\leadsto$  Daraus kann man schon DTMs für  $x := y + n$  erzeugen.
- Die Simulation von  $x := y - n$  ist analog möglich (mit zusätzlichem Test auf Gleichheit mit 0 beim Dekrementieren).
- Sequentielle Programmausführung  $P_1 ; P_2$  wird direkt im Zustandsgraphen der DTM umgesetzt („Hintereinanderhängen“ von TMs).
- While-Schleifen sind durch Zyklen im Zustandsgraphen darstellbar, wobei am Anfang jeweils ein Test auf Gleichheit mit 0 steht, um die Schleife verlassen zu können.

TM  $\rightarrow$  WHILE (1)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

# TM $\rightarrow$ WHILE (1)

## Behauptung 2: WHILE-Programme können DTMs simulieren:

- Wir nehmen zur Vereinfachung an, dass das TM-Arbeitsalphabet  $\Gamma = \{0, 1\}$  ist, und dass die Zustände natürliche Zahlen sind.
- Eine TM-Konfiguration  $a_1 a_2 \cdots a_p q a_{p+1} a_{p+2} \cdots a_\ell$  wird dargestellt durch drei Variablen:
  - `left` hat den Wert, der durch  $a_1 a_2 \cdots a_p$  binär kodiert wird (least significant bit ist dabei  $a_p$ );
  - `state` hat den Wert  $q$ ;
  - `thgir` hat den Wert, der durch  $a_\ell \cdots a_{p+2} a_{p+1}$  binär kodiert wird (least significant bit ist also  $a_{p+1}$ ).
- Diese Kodierung kann leicht auf größere Arbeitsalphabete erweitert werden ( $n$ -äre statt binäre Kodierung).

## TM $\rightarrow$ WHILE (2)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

- Wie gesagt:  
left hat den Wert, der durch  $a_1a_2 \cdots a_p$  binär kodiert wird

## TM $\rightarrow$ WHILE (2)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

- Wie gesagt:  
left hat den Wert, der durch  $a_1a_2 \cdots a_p$  binär kodiert wird
- Wir greifen auf (die Binärkodierung von) left wie auf einen Stapel (Keller, Stack) zu:

## TM $\rightarrow$ WHILE (2)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

- Wie gesagt:  
left hat den Wert, der durch  $a_1a_2 \cdots a_p$  binär kodiert wird
- Wir greifen auf (die Binärkodierung von) left wie auf einen Stapel (Keller, Stack) zu:
  - Pop: der folgende Pseudocode ist in WHILE (und LOOP) implementierbar

```
top := left mod 2  
left := left div 2
```

## TM $\rightarrow$ WHILE (2)

### Behauptung 2: WHILE-Programme können DTMs simulieren:

- Wie gesagt:  
left hat den Wert, der durch  $a_1a_2 \cdots a_p$  binär kodiert wird
- Wir greifen auf (die Binärkodierung von) left wie auf einen Stapel (Keller, Stack) zu:
  - Pop: der folgende Pseudocode ist in WHILE (und LOOP) implementierbar

```
top := left mod 2  
left := left div 2
```

- Push: der folgende Pseudocode ist in WHILE (und LOOP) implementierbar

```
left := left * 2 + top
```

- Auf thgir kann man genauso zugreifen.

## TM $\rightarrow$ WHILE (3)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

- Wir haben das Band in zwei Stacks kodiert, mit den Zeichen links und rechts neben dem TM-Kopf an oberster Stelle.

## TM $\rightarrow$ WHILE (3)

### Behauptung 2: WHILE-Programme können DTMs simulieren:

- Wir haben das Band in zwei Stacks kodiert, mit den Zeichen links und rechts neben dem TM-Kopf an oberster Stelle.
- Die TM-Simulation erfolgt jetzt in einer WHILE-Schleife:  
**WHILE** halt  $\neq 0$  **DO**  $P_{\text{Einzelschritt}}$  **END**
- Das Programm  $P_{\text{Einzelschritt}}$  führt einen Schritt aus:
  - `thgir.pop()` liefert das Zeichen an der Leseposition
  - Durch eine Folge von If-Bedingungen kann man für jede Kombination aus Zustand  $q$  (in state) und gelesenem Zeichen eine Behandlung festlegen
  - Schreiben von Symbol  $a$  durch `thgir.push(a)`
  - Bewegung nach rechts: `left.push(thgir.pop())`
  - Bewegung nach links: `thgir.push(left.pop())`
  - Zustandsänderung durch einfache Zuweisung
  - Anhalten durch Zuweisung `halt := 0`

## TM $\rightarrow$ WHILE (3)

**Behauptung 2:** WHILE-Programme können DTMs simulieren:

Zusammenfassung:

- Natürliche Zahlen simulieren Stacks der Bandsymbole links und rechts;
- Berechnungsschritte werden durch einfache Arithmetik implementiert (in LOOP möglich);
- eine WHILE-Schleife arbeitet die einzelnen Schritte ab, bis die TM hält.

Was fehlt noch zum detaillierten Beweis?

- Unsere Stack-Implementierung kann noch nicht mit dem leeren Stack umgehen.  
 $\leadsto$  Dies erfordert zusätzliche Tests und Sonderfälle (bei einseitig unendlichem TM-Band asymmetrisch).
- Für größere Arbeitsalphabete könnten wir statt Binärkodierung eine  $n$ -äre Kodierung verwenden. □

# Zusammenfassung und Ausblick

WHILE-Programme können alle berechenbaren Probleme lösen.  
(Dies ist ein weiteres Indiz für die Church-Turing-These.)

LOOP-Programme können fast alle praktisch relevanten Probleme lösen, aber nicht alle berechenbaren Probleme.

Beweistechniken: strukturelle Induktion, Widerspruch durch Selbstbezüglichkeit (Busy Beaver), TM mit einer While-Schleife und zwei Stacks simulieren

Was erwartet uns als nächstes?

- Relevantere Probleme
- Reduktionen
- Rice