# Berechenbarkeitstheorie

**David Basin**

Department of Computer Science
ETH Zurich

# Theoretische Informatik — Teil 2

## Willkommen!

- Dozent: David Basin
  - Für Fragen, Kommentare und Deutsch-Verbesserungen bin ich stets offen
  - Fragen sind willkommen: vor allem in der Pause
  - Gegenseitige Interaktion ist sehr erwünscht!
- Übungsbetrieb usw. bleibt unverändert
- Literatur: vorwiegend Uwe Schöning, Theoretische Informatik — kurzgefasst
- Folien stehen auf dem Web zur Verfügung

# Warum ist theoretische Informatik interessant?

- Grundlage für vieles!

  Algorithmen, Programmiersprachen, Kompilerbau, Verifikation, usw.

- Untersucht fundamentale Fragen
  - Welche Berechnungsprobleme kann man lösen?
  - Mit welchen Arten von Maschinen/Ressourcen/... ?

- Extrem praktisch und relevant!  Wie lösen Sie Probleme wie:
  - Kompileroptimierung: Eliminierung von unerreichbarem "Dead Code".
  - Reiseplanung (TSP): Minimiere Reisezeit des Aussendienstmitarbeiters

- Man lernt formales Denken

  Sehr wichtig: Programme, Systeme, ... sind formale Objekte!

- Die Materie (Ergebnisse & Beweismethoden) ist einfach schön!

# Überblick

1. **Berechenbarkeit**
   - Was kann man berechnen? D.h. welche mathematischen Funktionen können wir in endlicher Zeit berechnen?
   - Mit welcher Sprache/Maschine?
   - Welche Eigenschaften von Programmen sind entscheidbar? Z.B. Code-Erreichbarkeit.
2. **Komplexitätstheorie**
   - Was ist PTIME (= schnell) berechenbar?
   - Rolle von Nichtdeterminismus. P = NP?
   - Polynomial Space?
   - Einführung allgemeiner Techniken: Komplexitätsklassen, Reduktion, Härte-Probleme

# Berechenbarkeit — Intuitiv

- Eine (evtl. partielle) Funktion $f : N^k \to N$ ist berechenbar, falls es ein Rechenverfahren gibt, das $f$ berechnet.
  - Verfahren ist z.B. durch Java-Programm $P$ gegeben.
    Ausführung ohne Platzbeschränkungen
  - $P$ gestartet mit $(n_1, \ldots, n_k) \in N^k$ als Eingabe
    Terminiert nach endlich vielen Schritten mit Ausgabe $f(n_1, \ldots, n_k)$
  - Im Falle einer partiellen Funktion muss $P$ nicht (immer) stoppen

- **Beachten Sie**: Mathematisch gesehen ist eine Funktion eine Relation, eine Menge von geordneten Paaren. Z.B. $x \mapsto x^2$

$$\{(1,1),(2,4),(3,9),(4,16),(5,25),\ldots\}$$

Diese intuitive Definition setzt Funktionen mit Programmen in Beziehung.

# Beispiele

- Beispiel 1: Der Algorithmus

```
Input(n);
REPEAT UNTIL FALSE;
```

Definiert welche Funktion? Antwort: die total undefinierte Funktion ($\{\}$).

- Beispiel 2: Ist die Funktion

$$f(n) = \begin{cases} 1 & \text{falls } n \text{ ein Anfangsabschnitt der Dezimalbruchentwicklung von } \pi \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

berechenbar (z.B. f(314) $= 1$)?

Ja! Es gibt konvergierende Näherungsverfahren für die Zahl $\pi$.

# Beispiele (Forts.)

- Beispiel 3: Die Funktion

$$g(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der Dezimalbruchentwicklung von } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

  Berechenbarkeit ist nicht bekannt. Künftiges Wissen über $\pi$ reicht nicht aus, um eine Entscheidung über die Berechenbarkeit zu treffen.

- Möglicherweise könnte $g$ aus folgendem Grund berechenbar sein. Die Ziffernfolge von $\pi$ ist anscheinend zufällig, also kann es doch sein, dass *jede* Ziffernfolge irgendwann mal vorkommt. In diesem Fall wäre $g(n) = 1$ für alle $n$, welche eine (sehr einfache) berechenbare Funktion ist.

  **Beachten Sie**: Es könnte berechenbar sein, ohne dass wir es wissen!

# Beispiele (Forts.)

- Beispiel 4: Die Funktion

$$h(n) = \begin{cases} 1 & \text{falls in der Dezimalbruchentwicklung von } \pi \\ & \text{irgendwo mindestens } n\text{-mal hintereinander eine 7 vorkommt} \\ 0 & \text{sonst} \end{cases}$$

berechenbar! Entweder ergeben sich beliebig lange 7-er Sequenzen, dann ist $h(n) = 1$ für alle $n$. Oder es gibt eine $n_0$, so dass es 7-er Folgen nur bis zur Länge $n_0$ gibt, dann ist $h(n) = 1$, falls $n \leq n_0$ und $h(n) = 0$ sonst. In beiden Fällen ist $h$ berechenbar! Aber der Beweis dafür ist nicht konstruktiv (und muss nicht so sein).

# Berechenbarkeit — Churchsche These

- Berechenbarkeit hängt vom Begriff des verwendeten Rechenverfahrens ab.

- Es gibt viele solcher Begriffe

  Turingmaschinen, $\lambda$-kalkül, While-Programme (in Java, C, ...),
  Goto-Programme, $\mu$-rekursive Funktionen

- Erste Definitionen: Turing und Church, 1936.

  Es hat sich gezeigt, dass alle äquivalent sind!

- Heute ist man überzeugt, dass es keinen noch umfassenderen Begriff gibt.

  **Churchsche These:** Die durch die formale Definition der Turing-Berechenbarkeit (äquivalent: While, Goto, ...-Berechenbarkeit) erfasste Klasse von Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

# Churchsche These (Forts.)

- Kann die These bewiesen werden?

  Nein. “Intuitiver Sinn” ist nicht formal handhabbar.

- Kann die These benutzt werden?

  Sie wird oft benutzt, um zu argumentieren, dass ein Algorithmus existiert. Es wird dabei angenommen, dass der Algorithmus durch eine TM formalisiert werden kann. (In Übungen müssen Sie die Maschine explizit konstruieren!)

- Ist es überraschend?

# Turingmaschine

- Def. Eine Turingmaschine ist ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$$

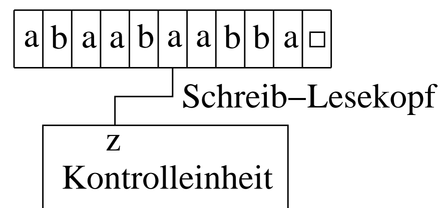
Hierbei sind:

- $Z$ die endliche Zustandsmenge,
- $\Sigma$ das Eingabealphabet,
- $\Gamma \supset \Sigma$ das Arbeitsalphabet,
- $\delta : Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ die Überführungsfunktion, im nichtdeterministischen Fall: $\delta : Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$
- $z_0 \in Z$ der Startzustand,
- $\Box \in \Gamma \setminus \Sigma$ das Blank,
- $E \subseteq Z$ die Menge der Endzustände.

Intuition: $\delta(z, a) = (z', b, x)$ bedeutet: Wenn sich $M$ in Zustand $z$ befindet und unter dem Schreib-/Lesekopf das Zeichen $a$ steht, so geht $M$ in den Zustand $z'$ über, schreibt (auf dem Platz von $a$) $b$ auf das Band und führt danach die Kopfbewegung $x \in \{L, R, N\}$ aus. Hier $L =$ links, $R =$ rechts und $N =$ neutral.

# Turingmaschine — Konfiguration

- Eine Konfiguration formalisiert den globalen Zustand der Turingmaschine.
  - Eine Konfiguration ist ein Wort $k \in \Gamma^* Z \Gamma^*$
  - $k = \alpha z \beta$ bedeutet, dass $\alpha\beta$ der nicht-leere Teil des Bandes ist. $z$ ist der Zustand, in dem sich die Maschine gerade befindet und der Schreib-/Lesekopf steht auf dem ersten Zeichen von $\beta$.
- Startkonfiguration mit Eingabe $x \in \Sigma^*$ ist $z_0 x$.
- Beispiel: abaabzaabba□




- Definiere erreichbare Konfigurationen durch binäre Relation $\vdash$

# Turingmaschine: $\vdash$

- $a_1 \ldots a_m z b_1 \ldots b_n \ \vdash$
  - $a_1 \ldots a_m z' c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, N)$, $m \geq 0$, $n \geq 1$.
  - $a_1 \ldots a_m c z' b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, R)$, $m \geq 0$, $n \geq 2$.
  - $a_1 \ldots a_{m-1} z' a_m c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, L)$, $m \geq 1$, $n \geq 1$.
- plus zwei Sonderfälle
  - $a_1 \ldots a_m z b_1 \vdash a_1 \ldots a_m c z' \Box$, falls $\delta(z, b_1) = (z', c, R)$
  - $z b_1 \ldots b_n \vdash z' \Box c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, L)$
- Eine TM kann sowohl Sprachen akzeptieren als auch Funktionen definieren.

# Turing-Berechenbarkeit und Definierbarkeit

- Def. Eine Turingmaschine akzeptiert die Sprache

$$T(M) = \{x \in \Sigma^* \mid z_0 x \vdash^* \alpha z \beta;\ \alpha, \beta \in \Gamma^*;\ z \in E\}$$

- Def. Eine Funktion $f : N^k \to N$ heißt Turing-berechenbar, falls es eine Turingmaschine $M$ gibt, so dass für alle $n_1, \ldots n_k, m \in N$ gilt

$$f(n_1, \ldots, n_k) = m \text{ gdw } z_o \overline{n_1} \# \overline{n_2} \# \ldots \# \overline{n_k} \vdash^* \Box \ldots \Box z_e \overline{m} \Box \ldots \Box$$

wobei $z_e \in E$ und $\overline{n}$ die (z.B. Binär-)Darstellung der Zahl $n$ darstellen.

- Def. Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt Turing-berechenbar, falls es eine Turingmaschine $M$ gibt, so dass für alle $x, y \in \Sigma^*$ gilt.

$$f(x) = y \text{ gdw } z_0 x \vdash^* \Box \ldots \Box z_e y \Box \ldots \Box$$

# Beispiel: unär +1

$$M = (\{z_0, z_e\}, \{1\}, \{1, \Box\}, \delta, z_0, \Box, \{z_e\})$$

wobei

$$\begin{aligned} \delta(z_0, \Box) &= (z_e, 1, N) \\ \delta(z_0, 1) &= (z_0, 1, R) \end{aligned}$$

- Beispiellauf: $z_0 11 \vdash 1 z_0 1 \vdash 11 z_0 \Box \vdash 11 z_e 1$
- Frage: welche Sprache wird akzeptiert?
  D.h. Was ist $T(M)$?

# Beispiel: binär +1

$$M = (\{z_0, z_1, z_2, z_e\}, \{0,1\}, \{0,1,\Box\}, \delta, z_0, \Box, \{z_e\})$$

wobei

$$\begin{array}{lllllllll}
\delta(z_0,0) & = & (z_0,0,R) & \delta(z_1,0) & = & (z_2,1,L) & \delta(z_2,0) & = & (z_2,0,L) \\
\delta(z_0,1) & = & (z_0,1,R) & \delta(z_1,1) & = & (z_1,0,L) & \delta(z_2,1) & = & (z_2,1,L) \\
\delta(z_0,\Box) & = & (z_1,\Box,L) & \delta(z_1,\Box) & = & (z_e,1,N) & \delta(z_2,\Box) & = & (z_e,\Box,R)
\end{array}$$

Beispiel:

$$z_0 101 \vdash 1 z_0 01 \vdash 10 z_0 1 \vdash 101 z_0 \Box \vdash 10 z_1 1 \Box$$
$$\vdash 1 z_1 00 \Box \vdash z_2 110 \Box \vdash z_2 \Box 110 \Box \vdash \Box z_e 110 \Box$$

# Beispiel

- Die überall undefinierte Funktion ist Turing-berechenbar:

$$\delta(z_0, a) = (z_0, a, R) \quad \text{für alle } a \in \Gamma$$

- Charakteristische Funktion:

  Man sagt, eine Sprache $A$ ist von Typ 0, wenn sie von einer Turingmaschine $M_A$ akzeptiert wird. Diese entspricht einer Turingmaschine, die die folgende Funktion $\chi_A : \Sigma^* \to \{0, 1\}$ berechnet

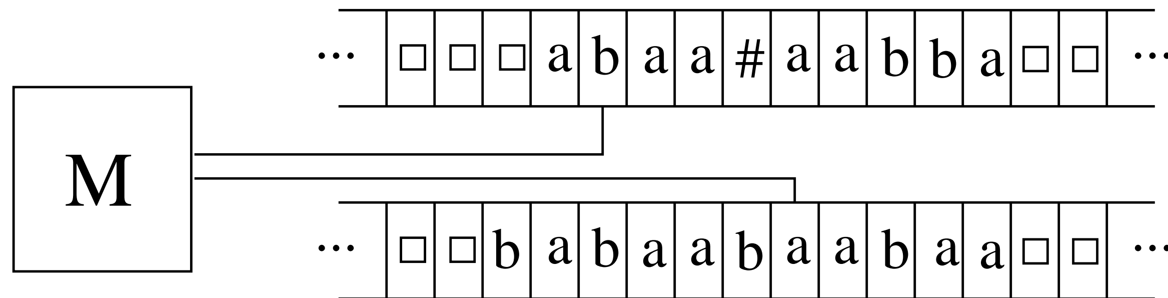
$$\chi_A(x) = \begin{cases} 1 & w \in A \\ \text{undefiniert} & w \notin A \end{cases}$$

$M_A$ kann leicht umgebaut werden, um $\chi_A$ zu berechnen. Daher stimmen die Typ 0-Sprachen genau mit den semi-entscheidbaren Sprachen überein.

# Mehrband-Turingmaschinen

- Es ist nützlich, Turingmaschinen mit weiteren “Fähigkeiten” zu erweitern, auch ohne eine nennenswerte Verbesserung der Mächtigkeit.

- Eine Mehrband-Turingmaschine kann auf $k \geq 1$ vielen Bändern unabhängig voneinander operieren. D.h. $k$ Schreib-Leseköpfe, und $\delta$ ist eine Funktion von $Z \times \Gamma^k$ nach $Z \times \Gamma^k \times \{L, R, N\}^k$




- Wir zeigen, dass diese TM keine zusätzliche “Berechnungskraft” besitzt.

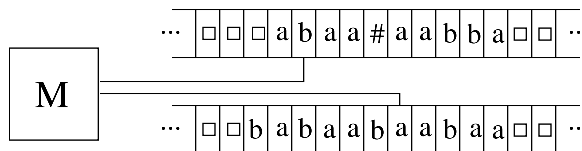
# Satz über Mehrband-Turingmaschinen

- Satz: Zu jeder Mehrband-Turingmaschine M gibt es eine (Einband-)Turingmaschine M' mit T(M) = T(M') bzw. so, dass $M'$ dieselbe Funktion berechnet wie $M$.

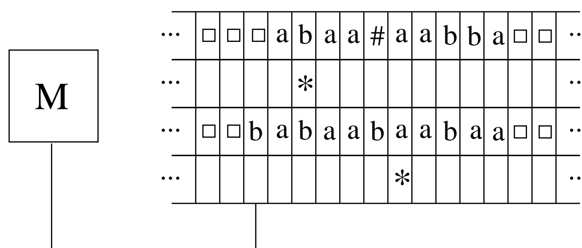
- Beweis

  Sei $k$ die Anzahl der Bänder von $M$ und $\Gamma$ das Arbeitsalphabet.

  Idee: Band von $M'$ in 2k-"Spuren" unterteilen, so dass eine Konfiguration von $M$ wie




  folgendermassen simuliert wird:



# Beweis (Forts.)

- Arbeitsalphabet von $M'$: $\Gamma' = \Gamma \cup (\Gamma \cup \{\star\})^{2k}$
- Simulation wie folgt:
  - Gestartet mit $a_1 a_2 \ldots a_n \in \Sigma^*$ erzeugt $M$ zunächst die Darstellung der Startkonfiguration von $M$ in der Spuren-Darstellung.
  - $M'$ simuliert einen $M$-Schritt durch mehrere Schritte:
    - $M'$ startet, so dass der Kopf links von allen $\star$-Markierungen steht
    - $M'$ geht nach rechts bis alle $k$ $\star$ überschritten werden
    - $M'$ hat jetzt alle Informationen, um die $\delta$-Funktionen von $M$ anzuwenden, d.h. die Eingabe $Z \times \Gamma^k$
    - $M'$ geht links über alle $\star$-Markierungen hinweg und führt alle entsprechenden Änderungen aus. QED
- N.B. gleiche Akzeptanz ist klar. Für gleiche Funktion müssen wir die Definition von Berechnung von Funktionen zu mehrbändigen TMs verallgemeinern. Z.B. Ausgabe auf dem ersten Band.

# TM: Ausdrucksmächtigkeit

- Man kann mehrbändige TM intuitiv programmieren.
- Notation: Sei $M$ eine 1-Band-TM. Wir schreiben $M(i,k)$, $i \leq k$ für die $k$-Band-TM, die wir aus $M$ erhalten, so dass die Aktionen von $M$ auf Band $i$ ablaufen und alle anderen Bänder unverändert bleiben.

  Z.B. $\delta(z,a) = (z', b, L)$ für $M$, dann sieht diese Transition in $M(3,5)$ etwa folgendermassen aus:

  $$\delta(z,(c_1,c_2,a,c_3,c_4)) = (z',(c_1,c_2,b,c_3,c_4),(N,N,L,N,N))$$

  Falls $k$ groß genug ist, schreiben wir nur $M(i)$ anstatt $M(i,k)$
- Die $+1$-(binär) Addier-TM bezeichnen wir mit $Band := Band + 1$

  Anstelle von "Band := Band+1" (i) schreiben wir "Band i := Band i + 1"
- Ähnlich können wir Mehrband-TMs erhalten, die die Operationen ausführen: Band $i$ := Band $i-1$, Band $i$ := 0 und Band $i$ := Band $j$.

# Ausdrucksmächtigkeit: Sequentielle Komposition

- Seien etwa $M_i = (Z_i, \Sigma, \Gamma_i, \delta_i, z_i, \Box, E_i)$, $i = 1, 2$. Wir bezeichnen die sequentielle Komposition von $M_1$ und $M_2$ durch (Flussdiagramm-Notation)

  $$\text{start} \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \text{stop}$$

  oder durch (Programmiersprachen-Notation) $M_1; M_2$.

- Dieses Hintereinanderschalten wird definiert durch

  $$M = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, z_1, \Box, E_2)$$

  wobei (oBdA) $Z_1 \cap Z_2 = \emptyset$ und

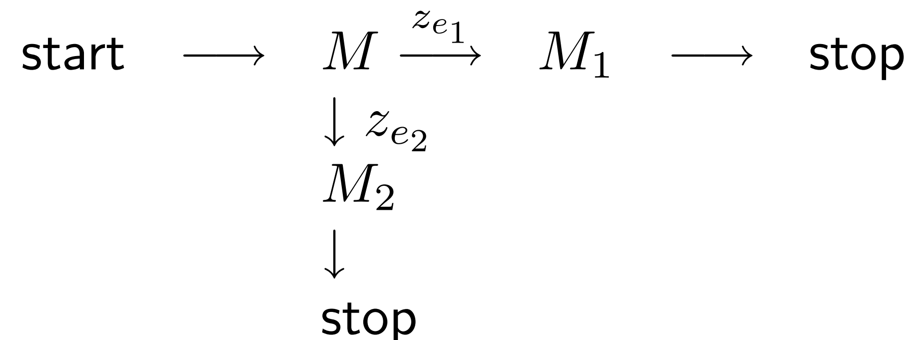
  $$\delta = \delta_1 \cup \delta_2 \cup \{(z_e, a, z_2, a, N) | z_e \in E_1, a \in \Gamma_1\}$$

- Beispiel: **start; Band := Band+1; Band := Band +1; Band := Band+1; stop**

  ist eine TM, die 3 hinzuaddiert.

# Ausdrucksmächtigkeit: if-then-else

- Nehmen wir an, dass $M$ mit "ja" (Zustand $z_{e_1}$) oder "nein" (Zustand $z_{e_2}$) terminiert
- In ähnlicher Weise bezeichnen wir mit




eine TM, wobei vom Endzustand $z_{e_1}$ von $M$ aus nach $M_1$ übergegangen wird, und von $z_{e_2}$ aus nach $M_2$. Alternative Schreibweise:

$$\texttt{if } M \texttt{ then } M_1 \texttt{ else } M_2$$

# Ausdrucksmächtigkeit: Test auf 0

- Als Beispiel einer derartigen “ja/nein”-$M$ (letzte Folie), betrachten wir eine TM, die wir mit “Band=0?” bezeichnen.

  $Z = \{z_0, z_1, ja, nein\}$; Startzustand $z_0$, Endzustände sind ja und nein.

$$
\begin{array}{rcl}
\delta(z_0, a) & = & (nein, a, N) \text{ für } a \neq 0 \\
\delta(z_0, 0) & = & (z_1, 0, R) \\
\delta(z_1, a) & = & (nein, a, L) \text{ für } a \neq \Box \\
\delta(z_1, \Box) & = & (ja, \Box, L)
\end{array}
$$

- Wie (nun) üblich schreiben wir “Band i =0?’ anstatt “Band=0?”(i).

# Ausdrucksmächtigkeit: Schleifen

- Schleife als Variante von if-then-else:

$$\begin{array}{lcl} \text{start} \longrightarrow & \text{Band } i = 0? & \xrightarrow{ja} \quad stop \\ & \uparrow\downarrow \; nein & \\ & M & \end{array}$$

- Alternative Schreibweise

$$\texttt{While Band } i \neq 0 \texttt{ DO } M$$

- Verschiedene, einfache programmiersprachen-ähnliche Konzepte können mit TMs simuliert werden.
  - Bandinhalte entsprechen Variablenwerten
  - Zuweisung, Hintereinanderreihung, Sprung und Schleifen

Läßt sich eine exakte Beziehung formal ausdrücken?

# LOOP-Berechenbarkeit

- Einfache Programmiersprache: genannt LOOP. Syntax:

  **Variablen:** $x_0, x_1, \ldots$
  **Konstanten:** 0, 1, 2, ...
  **Trennsymbole:** $:=$
  **Operationszeichen:** $+$, $-$
  **Schlüsselwörter:** LOOP DO END

- Syntax von LOOP-Programmen induktiv definiert:
  - Wertezuweisungen der Form $x_i := x_j + c$ bzw $x_i := x_j - c$, für $c \in N$, sind LOOP-Programme
  - Falls $P_1$ und $P_2$ LOOP-Programme sind, dann auch $P_1; P_2$
  - Falls $P$ ein LOOP-Programm ist und $x_i$ eine Variable, dann ist auch Loop $x_i$ DO $P$ END

# LOOP-Semantik

- Die Semantik von LOOP-Programmen ist wie folgt definiert:

  Für die Berechnung einer $k$-stelligen Funktion gehen wir davon aus, dass diese mit den Startwerten $n_1, \ldots, n_k \in N$ in den Variablen $x_1, \ldots x_k$ gestartet wird und alle anderen Variablen den Anfangswert 0 haben.

- Fälle:

  $\mathbf{x_i := x_j + c}$**:** Neuer Wert von $x_i$ ist Wert von $x_j$ plus Konstante $c$ (wie üblich)

  $\mathbf{x_i := x_j - c}$**:** "Proper Subtraction": falls $c > x_j$, so wird Resultat auf 0 gesetzt. Grund: Variablen dürfen nur Werte in $N$ haben.

  $\mathbf{P_1; P_2}$**:** führe zuerst $P_1$ und dann $P_2$ aus.

  **Loop** $\mathbf{x_i}$ **DO** $\mathbf{P}$ **END:** $P$ wird genau (der Anfangswert von) $x_i$-mal ausgeführt. N.B. das Ändern des Variablenwerts von $x_i$ im Inneren von $P$ hat keinen Einfluss auf die Anzahl der Wiederholung.

  Def. Eine Funktion $f : N^k \rightarrow N$ heißt LOOP-berechenbar, falls es ein LOOP-Programm $P$ gibt, das $f$ in dem Sinne berechnet, dass $P$, gestartet mit $n_1, \ldots n_k$ in den Variablen $x_1, \ldots x_k$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen $x_0$.

# LOOP (Forts.)

- Alle LOOP-berechenbaren Funktionen sind totale Funktionen.

  Warum? Sie terminieren nach endlich viel Zeit. Warum? Induktion über Programmaufbau (Annahme: Zuweisen/Testen braucht endlich viele Schritte).

- Sind alle totalen Funktionen LOOP-berechenbar?

  Nein. Aber Beispiele sind nicht so offensichtlich.

- N.B. spezielle Programmformen sind simulierbar (Zucker).
  - $x_i := x_j$ ist $x_i := x_j + 0$
  - $x_i := c$ ist $x_i = x_j + c$ für $x_j$ eine weiter nicht benutzte Variable, die noch den Anfangswert 0 hat
  - IF $x = 0$ THEN $A$ END simuliert als

    $y := 1$
    LOOP $x$ do y:= 0 END;
    LOOP $y$ DO $A$ END
  - If-then-else ähnlich formalisierbar

# LOOP Beispiel

- Beispiel: Addition

$$x_0 := x_1;$$
$$\text{LOOP } x_2 \text{ DO } x_0 := x_0 + 1 \text{ END}$$

Wir nennen dieses Programmstück $x_0 := x_1 + x_2$ und verallgemeinern dies auch auf beliebige andere Variablen-Indizes.

- Beispiel: Multiplikation (und ähnliche Verallgemeinerung)

$$x_0 := 0;$$
$$\text{LOOP } x_2 \text{ DO } x_0 := x_0 + x_1 \text{ END}$$

- Analog kann man MOD und DIV definieren. Wir erlauben uns LOOP-Programme mit komplizierten Wertzuweisungen, etwa

$$x := (y \ DIV \ z) + (x \ MOD \ 5) \times y$$

Dieses Programm kann in elementare Wertzuweisungen zerlegt werden.

# While-Programme

- Wir erweitern LOOP-Programme mit WHILE-Schleifen, und erhalten dadurch die WHILE-Programme.

  Def. Jedes LOOP-Programm ist ein WHILE-Programm und falls $P$ ein WHILE-Programm ist und $x_i$ eine Variable, dann ist auch

  $$\text{WHILE } x_i \neq 0 \text{ do } P \text{ END}$$

  ein WHILE-Programm.

- Die Semantik von WHILE ist so definiert, dass $P$ solange wiederholt auszuführen ist, wie der Wert von $x_i$ ungleich Null ist.

- N.B. Eine LOOP-Schleife kann immer mit einer WHILE-Schleife simuliert werden.

  $$\text{LOOP } x \text{ DO } P \text{ END}$$

  Simuliert durch folgendes ($y \notin var(P)$):

  $$y := x; \text{ while } y \neq 0 \text{ DO } y := y - 1;\ P \text{ END}$$

# WHILE-berechenbar

- Def. Eine Funktion $f : N^k \rightarrow N$ heißt WHILE-berechenbar, falls es ein WHILE-Programm $P$ gibt, das $f$ in dem Sinne berechnet, dass $P$, gestartet mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $f(n_1, \ldots, n_k)$ in der Variable $x_0$ — sofern $f(n_1, \ldots n_k)$ definiert ist, ansonsten stoppt $P$ nicht.

- Satz 1: Turingmaschinen können WHILE-Programme simulieren.

  Das heisst, jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

  Bew. Induktion über den Aufbau von WHILE-Programmen. Wir haben bereits gezeigt, wie Mehrband-TMs Wertzuweisungen, Sequenzenbildung und WHILE-Schleifen simulieren können. Schließlich kann noch jede Mehrband-TM wieder durch eine (1-Band-) TM simuliert werden.

# GOTO-Programme

- Wir wollen die Umkehrung zeigen: WHILE-Programme können TMs simulieren.

  Wir brauchen einen Zwischenschritt: GOTO-Programme

- Def. GOTO-Programme bestehen aus Sequenzen von Anweisungen $A_i$, die jeweils durch eine Marke $M_i$ eingeleitet werden:

$$M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_K$$

- Mögliche Anweisungen $A_i$ sind:

  **Wertzuweisungen:** $x_i := x_j + c$ (oder $x_i := x_j - c$)
  **Unbedingter Sprung:** GOTO $M_i$
  **Bedingter Sprung:** IF $x_i = c$ then GOTO $M_j$
  **Stopanweisung:** HALT

  Beim Schreiben von GOTO-Programmen lassen wir Marken, die niemals angesprungen werden können, oft weg.

- Die Semantik ist wie erwartet

# GOTO-Programme

- Satz 2: Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

  Bew. Eine WHILE-Schleife

  $$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

  wird simuliert durch

  $$\begin{array}{ll} M_1: & \text{IF } x_i = 0 \text{ THEN GOTO } M_2; \\ & P; \\ & \text{GOTO } M_1; \\ M_2: & \ldots \end{array}$$

- Diese Konzepte sind eigentlich äquivalent, d.h. die Umkehrung gilt auch.

# GOTO — Bew. von Umkehrung

- Satz 3: Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden

- Bew. Gegeben sei ein GOTO-Programm

$$M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_K$$

- Wir simulieren dies mit einem WHILE-Programm mit nur einer WHILE-Schleife

$$
\begin{array}{l}
count := 1; \\
\text{WHILE } count \neq 0 \text{ DO} \\
\quad \text{IF } count = 1 \text{ THEN } A'_1; \text{ END}; \\
\quad \text{IF } count = 2 \text{ THEN } A'_2; \text{ END}; \\
\quad\quad \vdots \\
\quad \text{IF } count = k \text{ THEN } A'_k; \text{ END};
\end{array}
$$

# GOTO — Bew. (Forts.)

- wobei $A'_i$ folgendermaßen definiert ist:

$A'_i$ =

| | |
|---|---|
| $x_j := x_l + c$; $count := count + 1$ | falls $A_i = x_j := x_l + c$ |
| $count := n$ | falls $A_i =$ GOTO $M_n$ |
| IF $x_j = c$ THEN $count := n$ ELSE $count := count + 1$ END | falls $A_i =$ IF $x_j = c$ THEN GOTO $M_n$ |
| $count := 0$ | falls $A_i =$ HALT |

- D.h. GOTO-Berechenbarkeit und WHILE-Berechenbarkeit sind äquivalent.

# Normalform für WHILE-Programme

- Aus dem Beweis kommt das folgende Korollar

  (Kleenesche Normalform für WHILE-Programme):

  Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.

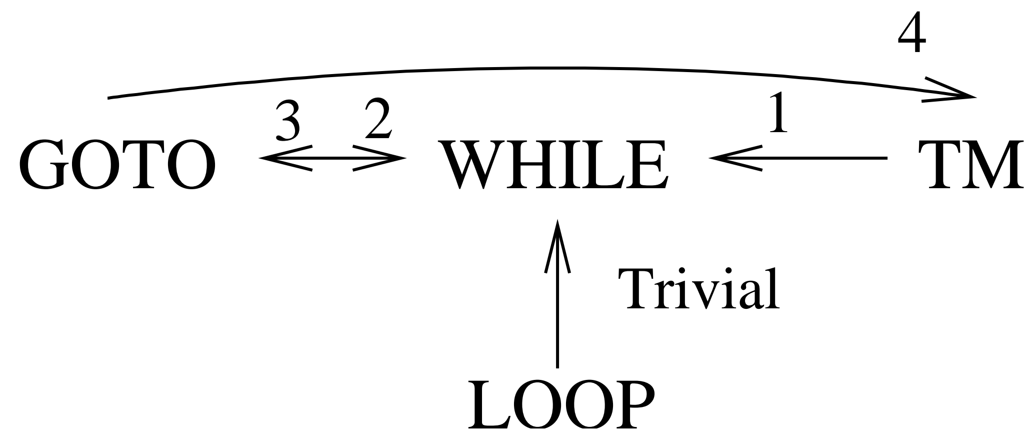
- Bew. Sei $P$ ein beliebiges WHILE-Programm. Wir formen $P$ zunächst um um ein äquivalentes GOTO-Programm $P'$ zu erhalten und dann wieder zurück in ein äquivalentes WHILE-Programm $P''$. Per Konstruktion hat dieses nur eine WHILE-Schleife.

# TMs und WHILE/GOTO-Programme

- Als nächstes zeigen wir:

  Satz 4: TMs können durch GOTO-Programme simuliert werden.

- Damit haben wir die folgenden Implikationen gezeigt:




D.h. GOTO-, WHILE- und TM-Berechenbarkeit sind äquivalent.

# Beweis Idee

- TM-Konfiguration

$$a_{i_1} \dots a_{i_p} z_l a_{j_1} \dots a_{j_q}$$

kann durch 3 (möglicherweise sehr grosse) Zahlen dargestellt werden $(x, y, z)$

$$\begin{array}{lllll} x & = & \overline{a_{i_1} \dots a_{i_p}} & = & (i_1 \dots i_p)_b \\ y & = & \overline{a_{j_q} \dots a_{j_1}} & = & (j_q \dots j_1)_b \\ z & = & \overline{z_l} & = & l \end{array}$$

wobei $(i_1 \dots i_p)_b$ die Zahl $i_1 \dots i_p$ in $b$-närer Darstellung bedeutet,

$$\Sigma_{u=1}^{p} i_u \times b^{p-u}$$

für $b$ gross genug ($b > |\Gamma|$).

- Man kann Tripel $(x, y, z)$ benutzen, um TM Schritt für Schritt zu simulieren.

  Da $y$ “rückwärts” repräsentiert, kann man mit DIV/MOD die Zellen direkt beim Lese-/Schreibkopf manipulieren.

# Beweis

- Sei $M = (Z, \Sigma, \Gamma, \delta, z_1, \square, E)$ eine TM zur Berechnung einer Funktion $f$.
- Wir simulieren $M$ durch ein GOTO-Programm mit folgender Struktur:

$$M_1 : P_1;\ M_2 : P_2;\ M_3 : P_3$$

  - $P_1$ transformiert Anfangswerte der Variablen in Binärdarstellung und erzeugt eine Darstellung der Startkonfiguration von $M$ in 3 Variablen $x$,$y$, $z$.
  - $P_2$ führt eine Schritt-für-Schritt-Simulation von $M$ mittels $x$, $y$, und $z$ durch.
  - $P_3$ erzeugt aus einer kodierten Endkonfiguration die Ausgabe in der Variablen $x_0$.
- N.B. $P_1$ und $P_3$ sind generische “Wrappings”. $P_2$ hängt von $M$’s Überführungsfunktion $\delta$ ab.

# Beweis (Forts.)

- Seien die Zustandsmenge $Z$ und Arbeitsalphabet $\Gamma$ durchnumeriert

$$Z = \{z_1, \ldots, z_k\} \quad \text{und} \quad \Gamma = \{a_1, \ldots, a_m\}$$

- Sei $b$ eine Zahl, $b > \Gamma$

- Wir benutzen die skizzierte Darstellung von $a_{i_1} \ldots a_{i_p} z_l a_{j_1} \ldots a_{j_q}$

$$\begin{aligned} x &= (i_1 \ldots i_p)_b \\ y &= (j_q \ldots j_1)_b \\ z &= l \end{aligned}$$

# GOTO-Programmstück für $P_2$

- Wir präsentieren die Konstruktion für $P_2$. ($P_1$ und $P_3$ sind einfache Aufgaben):
- $y\ MOD\ b$ ist $j_1$, d.h. der Index des Buchstaben, wo der Kopf steht. Also

```
M_2:    a := y MOD b;
        IF z = 1 AND a = 1 THEN GOTO M_11;
        IF z = 1 AND a = 2 THEN GOTO M_12;
        ⋮
        IF z = k AND a = m THEN GOTO M_km;
M_11 :  ⋆_11
        GOTO M_2;
M_12 :  ⋆_12
        GOTO M_2;
        ⋮
M_km :  ⋆_km
        GOTO M_2;
```

Die $M_{ij}$ für $1 \leq i \leq k$ und $1 \leq j \leq m$ stellen die Transition für $\delta(z_i, a_j)$ dar.

# $P_2$ (Forts.)

- In $M_{ij}$ nehmen wir an, dass die entsprechende $\delta$-Anweisung lautet:

$$\delta(z_i, a_j) = (z_{i'}, a_{j'}, L)$$

- Dies kann durch folgende Anweisungen ($\star_{ij}$) simuliert werden:

| | | | |
|---|---|---|---|
| $z$ | $:=$ | $i'$; | Zustand aktualisieren |
| $y$ | $:=$ | $y\ DIV\ b$; | Löschen von $a_j$ |
| $y$ | $:=$ | $b * y + j'$; | Schreiben von $a_{j'}$ |
| $y$ | $:=$ | $b * y + (x\ MOD\ b)$; | Bewegung links (I): $a_{i_p}$ jetzt vor Kopf |
| $x$ | $:=$ | $x\ DIV\ b$; | Bewegung links (II): $a_{i_p}$ entfernt von $x$ |

Andere Fälle sind ähnlich.

- Falls $z_i \in E$, beenden wir $\star_{ij}$ durch GOTO $M_3$. QED