“Theorie der Informatik” (CS206) Turingmaschine, LOOP-Programme, primitiv-rekursive Fkt, GOTO-Prog

25. März 2013

Proff Malte Helmert und Christian Tschudin

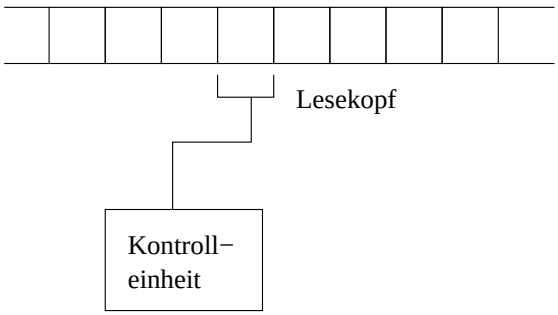
Departement Mathematik und Informatik, Universität Basel

Wiederholung/Einstieg

1. Was ist ein Keller-Automat (Push-Down Automaton)
2. Aus was besteht eine Turing-Maschine?
   Welcher Teil ist endlich, welcher nicht?
3. Was ist ein Transducer?
   Wie kann damit eine TM dargestellt werden?
4. Was ist die “Konfiguration” einer TM?

Definition Turing-Maschine (informell)

- **Unendliches** Speicherband, beweglicher Lesekopf
- **Endliche** Kontrolleinheit:
  – interner Zustand
  – Transitionsfunktion



TM – Terminologie (Konfiguration etc)

- Darstellung der “Ränder” des bisher benutzen Bandes:
  $\square \ldots \ Bandinhalt \ldots \ \square$
- Schreibweise für **“Konfiguration”**: $\alpha z \beta$
  – TM im Zustand z, Lesekopf steht auf erstem Symbol von β
- Startkonfiguration: $z_0 w \ \ (\rightarrow \alpha z_{end} \beta)$
  – TM im Zustand z_0, Lesekopf auf erstem Symbol von w
- “Zahlen”: z.B. Unärdarstellung: $\square \# xxx \ldots \ xx \# \square$
  – Anzahl x-Symbole hält die Zahl fest
  – Zahl wird durch $\#$ begrenzt

TM – Bezug zu “Sprache” und “Wortproblem”

- Als Wort dient die Start-Konfiguration
- Sprache T zur Maschine M:
  $$T(M) = \{a_1 \dots\ a_n \in \Sigma^* \mid z_0 a_1 \dots\ a_n \Rightarrow^* \alpha z_e\, \beta\}$$
  wobei $\alpha, \beta\ \in \Sigma^*, z_e \in E$
- **Wortproblem**
  ausgehend von allen möglichen Startkonfigurationen (Eingabewerte):
  Bei welchen hält die Maschine an? *(Dies definiert eine Sprache)*
  Oder auch: Wort von M akzeptiert $\Leftrightarrow$ M hält an

Mehrband–Turingmaschine

Einfacher, mit mehreren Bändern (=Variablen) zu arbeiten?

- TM-Maschine hätte mehrere Leseköpfe, die unabhängig voneinander bewegt werden können.
- Neue Transitions-Funktion:
  $\delta : Z\ \times\ \Gamma^k \rightarrow Z\ \times\ \Gamma^k\ \times \{L, R, N\}^k$
- **Satz:**
  Zu jeder Mehrband-TM M gibt es eine 1-Band-Maschine-TM M', die die selbe Sprache definiert ($T(M) = T(M')$).

D.h. wir könnten uns auf 1-Band-TM beschränken,
k-Band-Maschinen einsetzen falls nützlich.

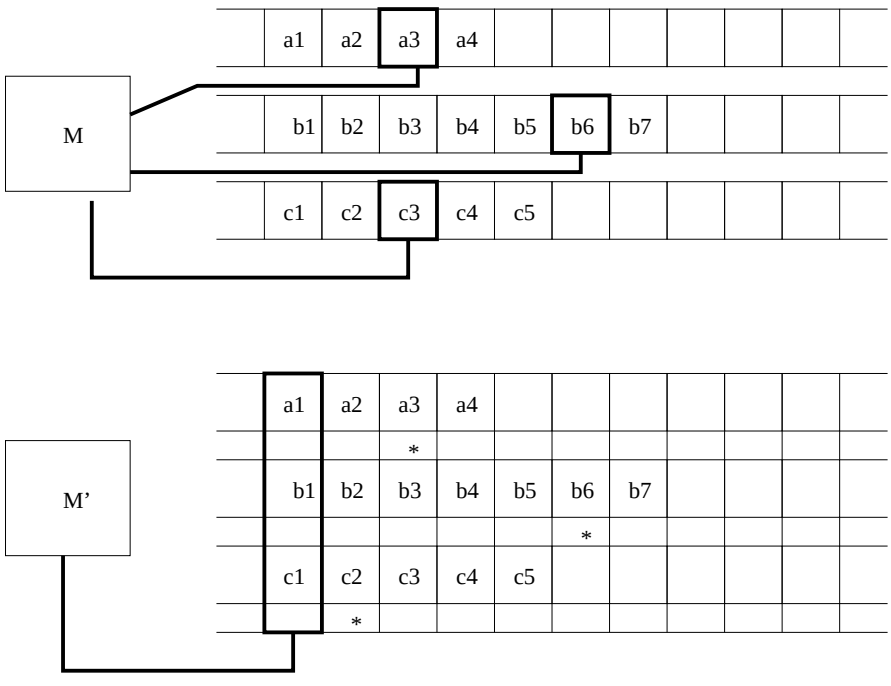
Grundidee der Abbildung Mehrband $\rightarrow$ 1-Band

Simulation (1-Band TM "berechnet" was die k-Band-TM macht)

- Neues (erweitertes) Arbeitsalphabet $\Gamma' = \Gamma \cup (\Gamma \cup \{*\})^{2k}$
  Stern markiert jeweile Lesekopf-Position
- D.h. neues Symbol ist "Vektor" aus alten Symbolen + Position
- Konkret: neues Arbeitsalphabet, dass alle möglichen Wert- und Position-Kombinationen abdeckt.

siehe Figur nächste Seite

Abbildung Mehrband $\rightarrow$ 1-Band, Schema



Grundidee der Abbildung Mehrband $\rightarrow$ 1-Band

Vorgehen:

- M' muss zuerste alle "Bänder" lesen, um Transition δ zu wählen
- Dazu:
  – positioniere Lesekopf links von allen Sternen $*$
  – Wischbewegung von links nach rechts bis alle Sterne gesehen
  – "notiere" (durch Zustandswechsel in M') alle mit Stern markierten Elemente
  – Bewegung von rechts nach links:
  wende Transition (gemäss alter Regel von M) an

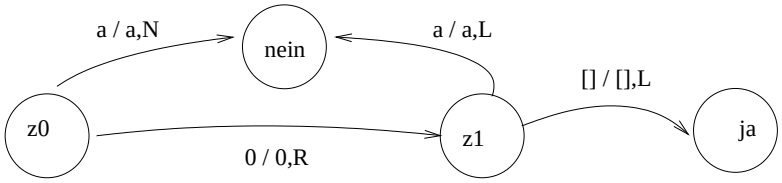
Verkettung von TMs

- Endzustand einer Maschine M_1 wird identifiziert mit Startzustand einer anderen Maschine M_2
- Konstruktion einer Gesamtmaschine
  – als Flussdiagramm: $start \rightarrow M_1 \rightarrow M_2 \rightarrow stop$
  – als sequentielles Programm: $M_1; M_2$
- Was wenn End-Zustandsmenge $|E| > 1$?
  Möglichkeit, dass TM M_1 zwei (oder mehr) verschiedenen Endzuständen hat:
  $\Rightarrow$ "Entweder-Oder-Maschine", Fortsetzung mit zwei verschiedenen Maschinen möglich

Eine “Test-auf-Null” Turing-Maschine

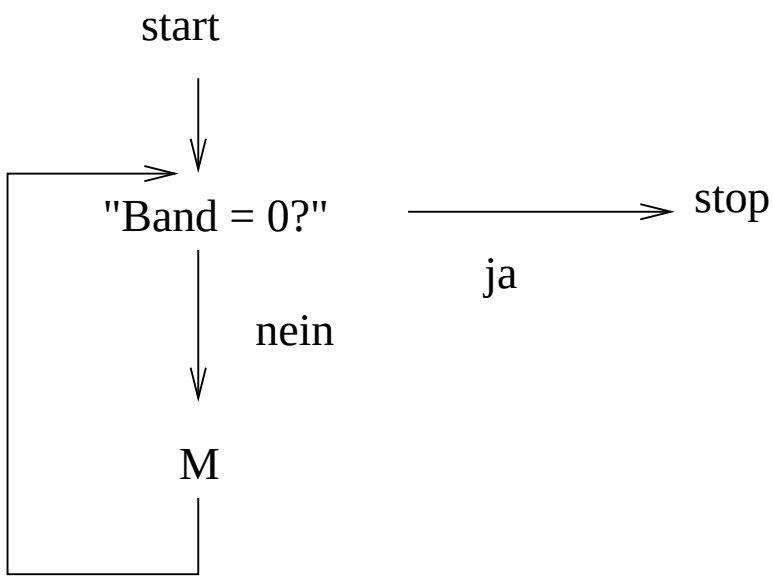
Zwei Endzustände $ja, nein$, wobei Endzustand ja angenommen wird, wenn ganz rechts eine Null (0) steht

```
delta(z0,a)  -> (nein,a,N)   // fuer alle a <> 0
delta(z0,0)  -> (z1,0,R)
delta(z1,a)  -> (nein,a,L)   // fuer alle a <> []
delta(z1,[]) -> (ja,[],L)
```



WHILE–Maschine

Jede TM M kann zu folgender Maschine erweitert werden: hält an, wenn das Resultat 0 ist.



WHILE-Maschine (Fortsetzung)

Als Programm: `WHILE (Band<>0) { M; }`

- WHILE-Programme entsprechen unserem “intuitiven” Berechenbarkeitsbegriff: **Alles kann durch ein (!) WHILE-Programm berechnet werden.**
- Einfachere (= weniger mächtigere) Programme: LOOP-Programme

Bevor wir die WHILE-Maschine weiterverfolgen, untersuchen wir LOOP-Programme und die Klasse der “primitiv-rekursiven Funktionen”

LOOP-Programme

Definition:

- Variablen x_0, ... x_j
- Konstanten 0, 1, 2
- Trennzeichen ; :=
- Operationen + -
- Schlüsselwörter LOOP DO OD

LOOP-Programme (Forts)

Rekursive Definition:

- Wertzuweisung:
  $x_i := x_j + c$
  $x_i := x_j - c$
- falls P1 und P2 ein LOOP-Programm ist, dann ist die Sequenz
  P1 ; P2
  auch ein LOOP-Programm
- Falls P ein LOOP-Programm und x eine Variable ist,
  dann ist LOOP x DO P OD
  auch ein LOOP-Programm

LOOP-Programme, Beispiele (Resultat in x0)

- Addition zweier Variablen:

```
ADD(x1, x2):
  x0 := x1
  LOOP x2 DO
    x0 := x0 + 1
  OD
```

- Multiplikation zweier Variablen:

```
MULT(x1, x2):
  x0 := 0
  LOOP x2 DO
    x0 := ADD(x0, x1)
  OD
```

LOOP-Programme, Beispiele (Forts)

- "if (x == 0) then A"

```
y := 1
LOOP x DO
  y := 0
OD
LOOP y DO
  A
OD
```

D.h. wir missbrauchen die numerische Variable y als "Flag", die Berechnung dieses Flags wird auch $testforzero(x)$ genannt.

LOOP-Programme und Funktionen

Welche "Funktionen" können mit LOOP-Programmen berechnet werden?

- Ziel des nun folgenden Abschnittes wird es sein, die LOOP-*Funktionklasse* zu charakterisieren.
- Vorgehen:
  – Definition der Klasse "primitiv-rekursiv"
  – später zeigen, dass dies der LOOP-Klasse entspricht

Primitiv-Rekursive Funktionen, Def

Basisfunktionen:

- Konstante Funktionen sind primitiv-rekursiv
- Projektionen sind primitiv-rekursiv, z.B.
  $\pi_2^3(a, b, c) == b$
  (im Exponent zu π steht die Stelligkeit, Index: Auswahl)
- Nachfolgefunktion auf $\mathbb{N}$ ist primitiv-rekursiv:
  $succ(n) == n + 1$

Fortsetzung auf der nächsten Seite...

Primitiv-Rekursive Fkt, Fortsetzung Def

Induktive Definition:

- Gleichzeitige **Substitution** (Komposition) von primitiv-rekursiven Funktionen ist auch primitiv-rekursiv, z.B.
  $f(x, y) = g(x, h(y))$
  (neue Funktion f wird aus g gebildet, wobei zweiter Parameter durch $h(y)$ ersetzt wird)
- Durch primitive **Rekursion** aus primitiv-rekursiven Funktionen definierte Funktion ist auch primitiv-rekursiv. Typisch:
  $f(0, ...) = g(...)$
  $f(n + 1, ...) = h(...f(n, ...), n, ...)$

Primitiv-Rekursive Funktionen

Beispiele von primitiv-rekursiven Funktionen für die schon gesehenen LOOP-Programme, sowie neue Funktionen

- $add(x, y)$
- $mult(x, y)$
- $testforzero(x)$
- Monus-Funktion
  $monus(x, y) == x - y$ falls $x \geq y$
  $monus(x, y) == 0$ falls $x < y$
- Vergleichsfunktionen ($>=$, $<$) und Minimum

Primitiv-Rekursive Funktionen, Beispiele 1

Wie sehen die Funktionen aus zu den LOOP-Programmbeispielen?

- Addition (Bemerkung: '+ mit Const' gehört zu LOOP-Programmen per Definition, die Addition selbst ist aber keine Basisfunktion primitiv-rekursiver Funktionen).

  Rekursive Definition der Funktion $add(a, b)$:
  $add(0, x) = x$
  $add(n + 1, x) = succ(add(n, x))$
- Vergleich mit Null (schon gesehen):
  $testforzero(0) = 1$
  $testforzero(x + 1) = 0$

Primitiv-Rekursive Funktionen, Beispiele 2

- Monus-Funktion
  Definition:
  $monus(x, y) == x - y$ falls $x \geq y$
  $monus(x, y) == 0$ falls $x < y$

- Vergleichsfunktionen:
  - $ge(x, y)$ (greater or equal, $\geq$)
    $ge(x, y) = testforzero(monus(y, x))$
  - $lt(x, y)$ (less than, $<$)
    $lt(x, y) = testforzero(ge(x, y))$

Primitiv-Rekursive Funktionen, Beispiele 3

- Minimum $min(x, y)$:
  $min(x, y) = mult(x, lt(x, y)) + mult(y, ge(x, y))$

Bezug LOOP-Programme $\Leftrightarrow$ prim-rek. Funktionen

Richtung $\Leftarrow$:
Zu jeder prim-rek Funktion gibt es ein LOOP-Programm

- Basisfunktionen sind klar, zB $succ(x)$:

```
resultat = x + 1;
```

- Komposition $f(x, y) = g(h(x, y), i(x, y))$
  wobei für g, h LOOP-Programme zur Verfügung stehen sollen:

```
a = h(x,y);
b = i(x,y);
resultat = g(a,b);
```

- Rekursion?

LOOP-Programm-Konstruktion für prim. Rekursion

Gegeben: $f(0, x_1, \ldots x_r) = g(x_1, \ldots x_r)$
$f(n+1, x_1, \ldots x_r) = h(f(n, x_1, \ldots x_r), n, x_1, \ldots x_r)$

Ausschreiben von $h(f(n, \ldots), \ldots)$:
$h(h(f(n-1, \ldots), n-1, \ldots), n, \ldots$

...

$h(h(\ldots\ h(f(0, x_1, \ldots x_r), 0, \ldots$

```
a = g(x1, ..., xr);
k = 0;
LOOP n DO
    a = h(a, k, x1, ..., xr);
    k = k + 1;
OD
```

Bezug LOOP-Programme $\Leftrightarrow$ prim-rek. Funktionen

Richtung $\Rightarrow$:
Zu jedem LOOP-Programm (für eine Funktion r) gibt es eine prim-rek Funktion R, die ebenfalls r berechnet.

- Problem:
  Was ist das "Ergebnis" der Ausführung eines LOOP-Programmes für eine Registermaschine? (Variabeln!)
- Ansatz: suche eine Abbildung
  Anfangskonfiguration $\rightarrow$ Zahl a
  Endkonfiguration $\rightarrow$ Zahl e
  so, dass $R(a) = e$ gilt.

Einschub: 2-Tupel auf $\mathbb{N}$ abbilden

Wie können **mehrere** Variablen-Werte auf **einen** Wert für die Anfangskonfiguration abgebildet werden?

Genereller Rahmen auch als "Gödelisierung" bekannt. Grundidee:

- Jeder (Turing-)Maschine T eine Zahl $n = G(T)$ zuordnen
- so dass aus einer gegebenen Gödelzahl n die Maschine $T = G^-1(n)$ rekonstruiert werden kann

Kodierung und Dekodierung kann mit LOOP-Programmen erfolgen!

Abbildung eines 2-Tupel auf $\mathbb{N}$

- Zähle alle Zahlenpaare auf, d.h. weise jedem Paar eine Nummer (Index) zu: $< x, y > \rightarrow c(x, y)$
- Geschickter Index (Funktion c(x,y)):

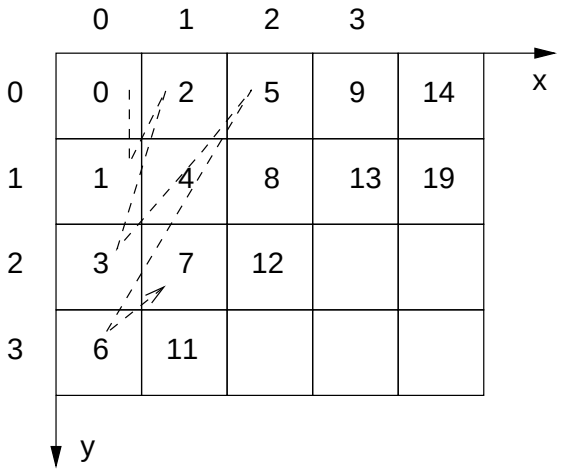
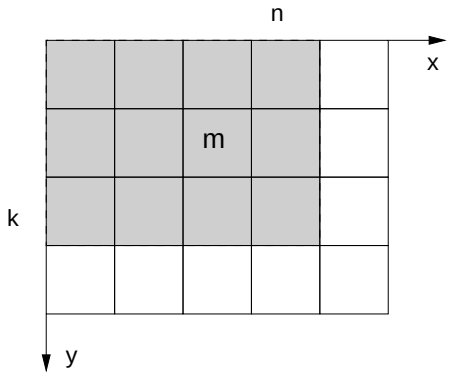


Abbildung eines 2-Tupel auf $\mathbb{N}$ (Fortsetz.)

- $c(x, y) = \binom{x + y + 1}{2} + x = \frac{(x + y + 1)(x + y)}{2} + x$
- Dies ist eine prim-rek. Funktion !
- Anwenden, um k-Tupel zu kodieren:
  $< n_0, n_1, \ldots, n_k > := \; c(n_0, c(n_1, \ldots \; c(n_k, 0)))$
- Nächstes Problem:
  Wie sehen die Umkehrfunktionen e und f von $c(x, y)$ aus?
  Wir wollen
  $e(c(x, y)) == x$
  $f(c(x, y)) == y$

Dekodierung eines gödelisierten Tupels

- Hilfsfunktion e'
  $e'(n, m, k) := max\{x \leq n | \ \exists y \leq k : \ c(x, y) = m\}$
  (Bestimme max Wert x im Feld $n \times k$ mit $c(x, y) = m$)



- Setze $e(m) := e'(m, m, m)$

Dekodierung (Forts)

- Analoge Funktion f' definieren für y,
  $f(n) = f'(n, n, n)$
- Mit $e(m)$ und $f(m)$ erhält man die Werte x, y zurück.
- Man kann zeigen, dass e' und f' prim-rek. sind.

Folgerung:
- Zu jedem LOOP-Programm eine Anfangs-Konfig berechnen
- Funktion zum Programm finden
- aus Endkonfig kann das Result des LOOP-Prog geholt werden,
- letzter Schritt ist auch primitiv-rekursiv.

Variante: Gödelisierung mit Primzahlen

Gegeben: Register-Werte $x_1, \ldots, x_r$
Gesucht Kodierung $K(x_1, \ldots, x_r) \in \mathbb{N}$, invertierbar

Kodierung:
$K^k(x_1, \ldots, x_r) := \prod_{i \leq k} p(i)^{x_i}$, wobei $p(i)$ die i-te Primzahl ist

- Beispiel: $x_1 = 1, x_2 = 3, x_3 = 0, x_4 = 5$
  $p(1)^1 \times p(2)^3 \times p(3)^0 \times p(4)^5 = \ldots = 907578$

Dekodierung: (aus Gödelzahl n die i-te Komponente extrahieren)
$D(n, i) := max\{j | n \ \ mod \ \ p(i)^j = 0\}$

LOOP-Programme $\Leftrightarrow$ prim-rek. Funktionen

Satz

Die Klasse der primitiv rekursiven Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein.

Bemerkungen:

- LOOP-Programme sind total, d.h. unabhängig von der initialen Belegung der Variablen bricht ein LOOP-Programm immer ab.
- Es gibt (intuitiv) berechenbare Funktionen, die nicht prim-rek. sind. ($\rightarrow$ WHILE-Programme, GOTO-Programme, volle TM)

Ackermann-Funktion

Die Ackermann-Funktion war eines der ersten Beispiele (1928) einer arithmetischen Funktion, die *nicht* LOOP-berechenbar ist.

Rekursive Definition:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Siehe Demo: viele Berechnungsschritte, *sehr* schnelles Wachstum

Ackermann-Funktion (Forts)

m\n	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2$
2	3	5	7	9	11	$2n+3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$A(3, A(4, 3))$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ twos}}-3$
5	65533	$A(4, 65533)$	$A(4, A(5, 1))$	$A(4, A(5, 2))$	$A(4, A(5, 3))$	$A(4, A(5, n-1))$
6	$A(5, 1)$	$A(5, A(5, 1))$	$A(5, A(6, 1))$	$A(5, A(6, 2))$	$A(5, A(6, 3))$	$A(5, A(6, n-1))$

Uebersicht

Bisher:

- WHILE-Maschine, WHILE-Programm
- LOOP-Programme,
  und primitiv-rekursive Funktionen

Folgend:

- GOTO-Programme, Aequivalenz zu WHILE-Programmen

Anschliessend:
μ-Rekursion, Aequivalenz zu WHILE-Programmen

GOTO-Programm – Definition

Grundidee: Programmierung einer einfachen (Register)-CPU

- **Programm = Sequenz von Zeilen** mit
  `MARKE : ANWEISUNG`
- Anweisungen sind:
  - Wert-Zuweisung x_i := x_j + c
  - unbedingter Sprung: GOTO M_i
  - bedingter Sprung: IF x_i == c THEN GOTO M_i
  - Stopanweisung: HALT

Bemerkung – Möglichkeit unendlicher Schleifen: `M1 : GOTO M1`

GOTO-Programm – Beispiel

Berechnung der Fakultätsfunktion $n!$

```
M0: n = ...    # Parameter n
M1: f = 1
M2: IF (n <= 1) THEN GOTO M6
M3: f = f * n
M4: n = n - 1
M5: GOTO M2
M6: HALT       # Resultat f
```

```
    n = ...
    f = 1
L1: IF (n <= 1) THEN GOTO L2
    f = f * n
    n = n - 1
    GOTO L1
L2: HALT
```

(fakultät() kann durchaus als LOOP-Program formuliert werden)

WHILE- und GOTO-Programme

- Jedes GOTO-Programm kann als WHILE-Programm umgeschrieben werden
- Jedes WHILE-Programm kann als GOTO-Programm umgeschrieben werden

Ein WHILE-Prog in ein GOTO-Prog umformen

Aus

```
WHILE x <> 0 DO  P  OD
```

wird

```
M1 : IF x = 0 THEN GOTO M4
M2 : P
M3 : GOTO M1
M4 : ...
```


Ein GOTO-Prog in ein WHILE-Prog umformen

Ein GOTO-Programm

```
M1 : A1;    M2 : A2;  ...   Mk : Ak
```

wird abgearbeitet durch

```
count = 1
WHILE count <> 0 DO
  IF count = 1 THEN A1';
  IF count = 2 THEN A2';
  ...
OD
```

wobei A_i' wie folgt definiert ist (siehe nächstes Slide)

GOTO $\rightarrow$ WHILE (Forts)

A_i' ist wie folgt definiert:

- falls A_i die Form $x_j = x_l + c$ hat:
  `x`$_j$ `= x`$_l$ `+ c; count = count + 1`
- falls A_i die Form $GOTO\ M_n$ hat:
  `count = n`
- falls A_i die Form $IF\ x_j = c\ THEN\ GOTO\ M_n$ hat:
  `IF x`$_j$`=c THEN count=n ELSE count=count+1`
- falls A_i die Form $HALT$ hat:
  `count = 0`

Stellenwert von GOTO für die Programmierpraxis

- Heute gängige CPUs sind GOTO-Maschinen:
  – Register (inkl Hauptspeicher)
  – sequenzielles Abarbeiten von Maschinenbefehlen (Marke = Speicheradresse)
  – explizite Sprungbefehle: `jump, jump-conditional`
- GOTO ist heute auf Assembler-Bereich beschränkt
  – Programmier-Hochsprachen vermeiden GOTOs (siehe nächstes Slide)
  – GOTO ist in C noch möglich ...

Stellenwert von GOTO (Fortsetzung)

Edsger W. Dijkstra: ***Go To Statement Considered Harmful*** (CACM 1968)

- Entwurf von Programmiersprachen bringt Einsicht, dass GOTO zu schlechten Programmen führt. ($\rightarrow$ "strukturiertes Programme" statt "Spaghetti-Code")
- Titel des Beitrags ist in Computer Science zu einer Redewendung geworden: *"XYZ Considered Harmful"*

<< For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so. >>

Uebersicht zum Abschluss für heute:

- LOOP $\approx$ primitiv-rekursiv
- LOOP ist weniger mächtig als WHILE
- GOTO $\approx$ WHILE $\approx$??
  $\Rightarrow \mu$-Rekursion (siehe nächste Sitzung)