Theoretische Informatik

Berechenbarkeit

Inhalt

1. Turingmaschinen
   - Definition und Beispiele
   - Turing-Post Programme
   - Registermaschinen
   - Äquivalenz
2. Programmierparadigmen
   - GOTO
   - LOOP- und WHILE-Sprachen
   - Äquivalenz von GOTO und WHILE
   - Churchsche These
3. Rekursive Funktionen
   - Primitive Rekursion
   - Äquivalenz zu LOOP
   - µ-Rekursion
   - Äquivalenz zu While
4. Halteproblem
   - Aufzählbarkeit u. Entscheidbarkeit
   - Halteproblem
   - Nichtberechenbare Funktionen
   - Satz von Rice

© H. Peter Gumm, Philipps-Universität Marburg

Inhalt

1. Turingmaschinen
   - Definition und Beispiele
   - Turing-Post Programme
   - Registermaschinen
   - Äquivalenz
2. Programmierparadigmen
   - GOTO
   - LOOP- und WHILE-Sprachen
   - Äquivalenz von GOTO und WHILE
   - Churchsche These
3. Rekursive Funktionen
   - Primitive Rekursion
   - Äquivalenz zu LOOP
   - μ-Rekursion
   - Äquivalenz zu While
4. Halteproblem
   - Aufzählbarkeit u. Entscheidbarkeit
   - Halteproblem
   - Nichtberechenbare Funktionen
   - Satz von Rice

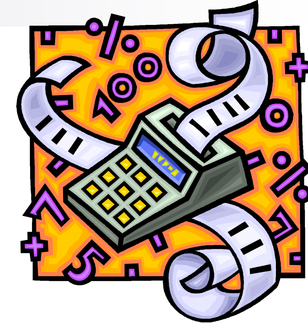
© H. Peter Gumm, Philipps-Universität Marburg

Algorithmus

- Ein Algorithmus ist eine detaillierte und unzweideutige Vorschrift zur schrittweisen Lösung einer Aufgabe
- Was bedeuten die Begriffe ?
  - Detailliert
    - Alle Fragen zur Ausführung müssen eindeutig beantwortet sein.
    - Alle Eventualitäten berücksichtigt
  - Schrittweise
    - Ausführung geschieht in Einzelschritten
    - Im Gegensatz zu kontinuierlichen Prozessen
  - Unzweideutig
    - Die erlaubten Aktionen müssen klar definiert sein
    - Bedingungen für die Anwendung eines Schrittes müssen stets klar sein
- Algorithmus muss von einem Menschen oder einer Maschine ausgeführt werden können.
- Das Ergebnis muss bei gleichen Anfangsbedingungen gleich ausfallen

© H. Peter Gumm, Philipps-Universität Marburg

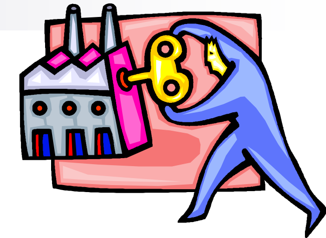
Berechenbare Funktionen



Seine Σ und Γ Alphabete

- Eine Funktion $f : \Sigma^* \rightarrow \Gamma^*$ heißt berechenbar, wenn es einen Algorithmus gibt, der sie berechnet
- Fragen:
  - Gibt es Funktionen, die nicht berechenbar sind ?
  - Was ist mit Funktionen $f: N^k \rightarrow N$?
  - Was ist mit nicht überall definierten Funktionen ?

Codierungen



- Jede Zahl $n \in N$ kann als Wort über $\Sigma = \{0,1\}$ aufgefasst werden
- Umgekehrt kann jedes Wort $w \in \Sigma^*$ als natürliche Zahl aufgefasst werden:
  - Für $|\Sigma| = n$ kann man Zeichen aus Σ als Ziffern im n-System deuten
- Kleine Komplikation:
  - $001 \neq 01$ verschieden als Worte von $\{0,1\}^*$, aber gleich als Zahlen: $(001)_2 = (01)_2$
- Bijektive Codierungsfunktion $c:\{0,1\}^* \to N$ durch:

$$c(w)=(1w)_2-1$$

  mit Dekodierfunktion $d:N \to \{0,1\}^*$ gegeben durch

$$d(n)= d_1 d_2 \ldots d_k \in \{0,1\}^*$$

  wobei $(1\, d_1 \ldots d_k)_2 = n+1$

- Es gilt $d \circ c = id_{\{0,1\}^*}$ und $c \circ d = id_N$, also beide eindeutig umkehrbar

© H. Peter Gumm, Philipps-Universität Marburg

Berechenbare Funktionen

- Eine Funktion $f : N^k \rightarrow N$ heißt berechenbar, falls es einen Algorithmus gibt, der diese Funktion berechnet.
  - Äquivalentes Konzept zu dem vorigen
  - Mittels Codierung kann man Funktionen

    $f:\Sigma^* \rightarrow \Gamma^*$

    durch Funktionen

    $f': N \rightarrow N$ bzw. $f'':N^k \rightarrow N^r$

    ersetzen und umgekehrt
  - Die Codierungen und Dekodierungen sind intuitiv berechenbar.
- $f : N^k \rightarrow N^r$ heißt berechenbar, falls die Koordinatenfunktion $\pi_i \circ f$ für alle $i \leq n$ berechenbar sind

© H. Peter Gumm, Philipps-Universität Marburg

Beispiele

- Die folgenden Funktionen sind berechenbar.
  - Wir benutzen Java als Algorithmenkonzept
  - Da Java keinen Datentyp **`nat`** besitzt benutzen wir **`int`**:

```
int pair(int x, int y){
    return (x+y)*(x+y+1)/2+x;
}
int maxGauss(int k){
    int i=0;
    while (i*(i+1)/2 <= k)
    i++;
    return --i;
}
```

```
int pi1(int k){
    int i=maxGauss(k);
    return k-i*(i+1)/2;
}

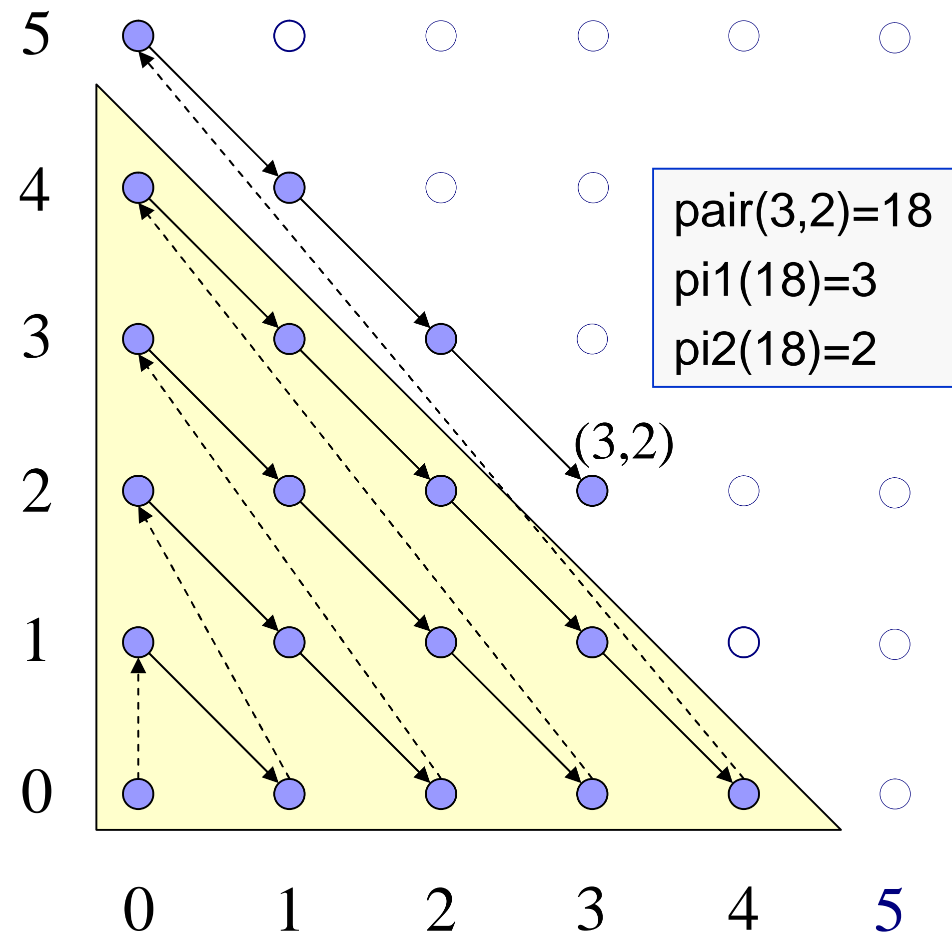
int pi2(int k){
    int i=maxGauss(k);
    return (i+1)*(i+2)/2-k-1;
}
```

© H. Peter Gumm, Philipps-Universität Marburg

Berechenbare Bijektion N×N → N

- Die Funktion pair ist eine Bijektion zwischen N×N und N.
- Umkehrfunktion N → N×N ist gegeben durch
  f(n) = (pi1(n),pi2(n))
- Das gelbe Dreieck hat gerade 1+2+...+ i = i*(i+1)/2 viele Punkte, wobei

  i=maxGauss(n)



© H. Peter Gumm, Philipps-Universität Marburg

Berechenbare Bijektion N*→N

Wir identifizieren hier N^* mit der Menge aller endlichen Listen von Elementen aus N, also

[] = leere Liste, und
$[n_1,\ldots,n_k]$ Liste der Zahlen $n_1,\ldots,n_k$

- $f([]) = 0$ // leere Liste
- $f([n_1]) = 1+pair(0,n_1)$ // Einerliste
- $f([n_1,\ldots,n_k]) = 1 + pair(k-1,pair(n_1,\ldots,pair(n_{k-1},n_k) \ldots)$ // allgemein
  - eindeutig dekodierbar
  - surjektiv
- Beweis: Übung

© H. Peter Gumm, Philipps-Universität Marburg

Partielle berechenbare Funktionen

- Algorithmen kann man evtl. nicht ansehen, für welche Inputs sie terminieren. Daher lässt man partielle Funktionen zu, deren Definitionsbereich eine Teilmenge von N^k ist.

```
int ulam(int n){
   while(n>1){
     if (n%2==0) n=n/2
     else n=(3*n+1)/2
   return 1;
}
```

Keine Ahnung, für welche n **ulam** terminiert

Vermutung: Für alle $n \in N$

Bis heute unbewiesen.

Generell nennt man f eine partielle Funktion und schreibt

$$f :: N^k \to N,$$

falls f auf einer Teilmenge U von N^k definiert ist. U heißt domain von f, kurz: dom(f). Schreibweise:

$$f(n) = \bot \quad \text{für} \quad n \notin dom(f)$$

© H. Peter Gumm, Philipps-Universität Marburg

Beispiele partieller berechenbarer Funktionen

- Eine nur auf den geraden Zahlen definierte Funktion
  - $f(n) = \begin{cases} n/2, & \text{falls } n \text{ gerade} \\ \bot & \text{, sonst} \end{cases}$

```
int f(int n){
   while(n%2!=0){};
   return n/2;
}
```

- Die überall undefinierte Funktion Ω:
  - $\Omega(n) = \bot$

```
int omega(int n){
   while(true){};
   return 0;
}
```

© H. Peter Gumm, Philipps-Universität Marburg

Dubiose Konsequenz klassischer Logik

- Primzahlzwillinge (PZZ) sind Paare (p, p+2), so dass p und p+2 prim sind
  - Beispiele: (3,5), (11,13), (17,19), (41,43)
  - Unbekannt: Gibt es unendlich viele PZZ?

- Definiere eine Funktion:
  - $f(n)=\begin{cases} 1, \text{ falls es unendl. viele PZZ gibt} \\ 0, \text{ sonst} \end{cases}$

  - Ist f berechenbar ? Ja.
  - Grund: Dubioses Axiom der Mathematik: *Tertium non datur*

- Konsequenz: Algorithmus ist einer der folgenden zwei:

```
int f(int n){
        return 1;
}
```

```
int f(int n){
        return 0;
}
```

- Welcher ? Wenn ich das wüsste ...

© H. Peter Gumm, Philipps-Universität Marburg

Diagonalisierung

Georg Cantor
1845 - 1918

- G. Cantor: Es gibt keine bijektive Abbildung zwischen N und $\wp(N)$

- Angenommen $\Phi : N \rightarrow \wp(N)$ sei surjektiv. Betrachte

$$D := \{ n \in N \mid n \notin \Phi(n) \}$$

$D \in \wp(N)$, also muss ein $d \in N$ existieren mit $\Phi(d) = D$.

Frage jetzt:

Ist $d \in D$?

Antwort:

$$d \in D \iff d \notin \Phi(d) \iff d \notin D.$$

Widerspruch !

$\wp(X)$

© H. Peter Gumm, Philipps-Universität Marburg

Diagonalisierung

Georg Cantor
1845 - 1918

- G. Cantor: Es gibt keine bijektive Abbildung zwischen N und ℘(N)

- Angenommen $\Phi : N \rightarrow \wp(N)$ sei surjektiv. Betrachte

$$D := \{ n \in N \mid n \notin \Phi(n) \}$$

$D \in \wp(N)$, also muss ein $d \in N$ existieren mit $\Phi(d)=D$.

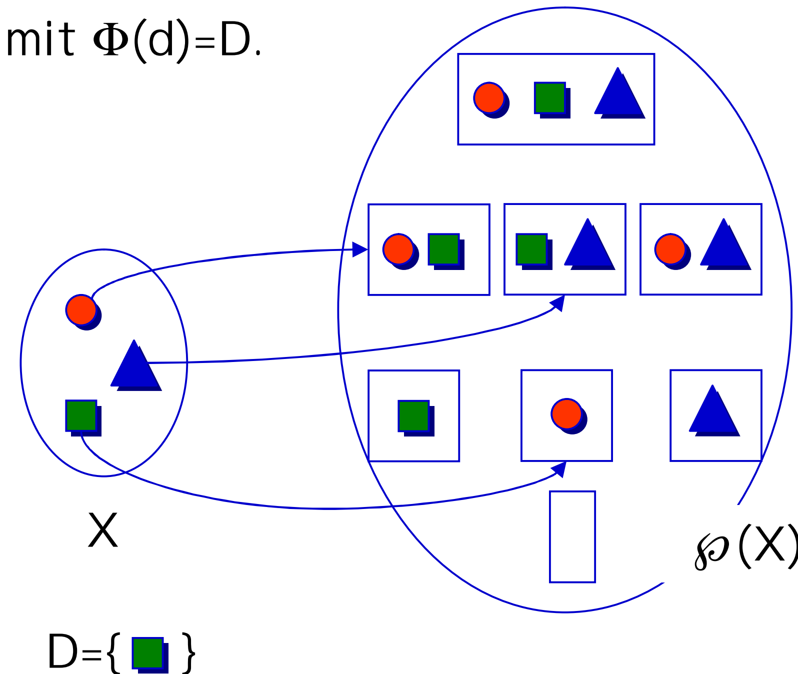
Frage jetzt:

Ist $d \in D$?

Antwort:

$$d \in D \Leftrightarrow d \notin \Phi(d) \Leftrightarrow d \notin D.$$

Widerspruch !



© H. Peter Gumm, Philipps-Universität Marburg

Geometrische Veranschaulichung

- Ersetze Teilmengen U von N durch ihre charakteristischen Funktionen $\chi_U : N \to Bool=\{0,1\}$
  - $\chi_U(n) = \begin{cases} 1, \text{ falls } n \in U \\ 0, \text{ sonst} \end{cases}$

Egal wie $\Phi : N \to \wp(N)$ definiert ist, finden wir eine Menge D, die nicht im Bild von Φ sein kann:

Für jedes n unterscheiden sich D und $\Phi(n)$:

$n \in D \Leftrightarrow n \notin \Phi(n)$

	0	1	2	3	4	5	6	7	8	9	…	
$\Phi(0)$	**0**	1	1	0	0	0	1	1	0	1	1	1
$\Phi(1)$	1	**1**	0	1	1	1	1	0	1	1	0	1
$\Phi(2)$	1	0	**1**	1	0	1	1	0	0	0	0	1
$\Phi(3)$	0	0	0	**0**	0	0	1	1	0	0	0	1
$\Phi(4)$	1	1	1	0	**1**	1	0	1	1	0	0	0
$\Phi(5)$	1	1	1	0	0	**0**	1	0	1	0	1	0

Eine Zeile, die nirgends in der Tabelle vorkommt …

D	**1**	**0**	**0**	**1**	**0**	**1**	…	…	…	…	…	…

© H. Peter Gumm, Philipps-Universität Marburg

Existenz nicht berechenbarer Funktionen

- Cantor:
  - Es gibt mehr Teilmengen von N als Elemente von N
  - ⇒ Es gibt mehr als |N| viele Funktionen $f : N \to N$
- Andererseits:
  - Σ^* hat höchstens |N| viele Elemente
  - Jeder Algorithmus ist ein Element aus Σ^*
    - z.B. Jedes Java-Programm ist $\in$ UNICODE*
  - ⇒ Es gibt höchstens |N| viele berechenbare Funktionen
- Konsequenz:
  - Es gibt Funktionen $f : N \to N$, die nicht berechenbar sind

© H. Peter Gumm, Philipps-Universität Marburg

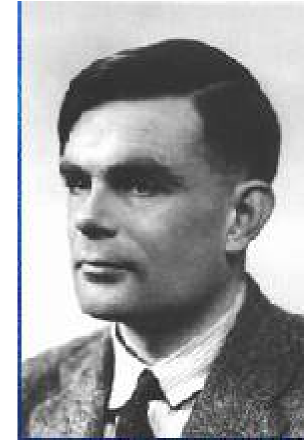
Zeig mir eine nicht berechenbare Funktion

- Die Existenz nicht berechenbarer Funktionen ist nur durch Widerspruchsbeweis geführt.
  - Kann man eine konkrete Funktion vorzeigen ?
- Gesucht also
  - eine Funktion f : N→N und
  - ein Beweis, dass es keinen Algorithmus zur Berechnung von f gibt
- Dazu müssen wir den Begriff „Algorithmus“ mathematisch präzise fassen.
  - Aber wie ?

© H. Peter Gumm, Philipps-Universität Marburg

Alan Turing

- Historisch erste mathematische Präzisierung der Begriffe
  - Algorithmus
  - partielle berechenbare Funktion
- Algorithmus als Maschine
  - Vorbild: Mensch mit kariertem Papier („Computer")
    - Schreibt was in Kästchen
    - Radiert
    - Macht Zwischenrechnungen und speichert diese
  - Abstrahiere zu möglichst einfachem Modell
    - 2-dim. Rechenblatt à 1-dimensionales Band
    - Lese und schreibe jederzeit nur auf einem Kästchen
    - Endlich viele Zustände
      - entsprechen Phase der Berechnung
- Ergebnis: Turingmaschine

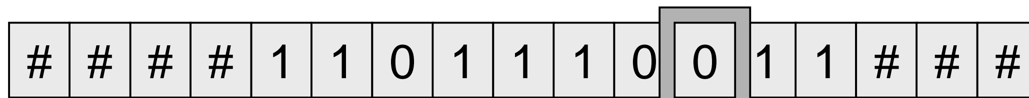


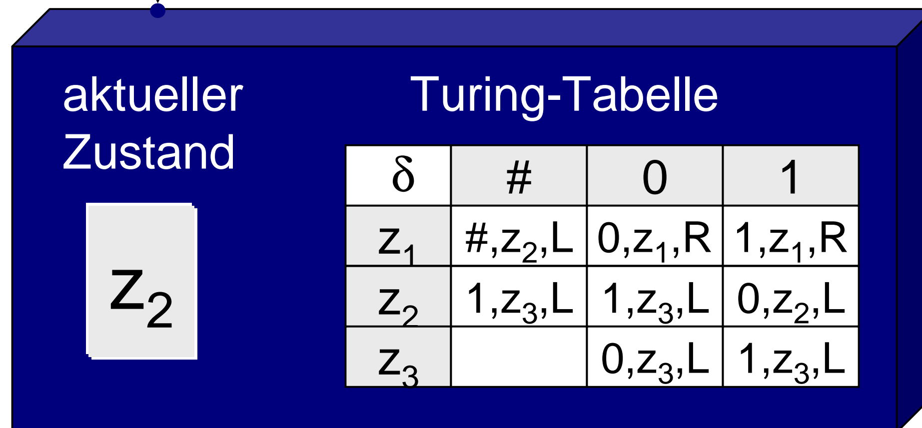
Alan Turing

1912-1954

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine





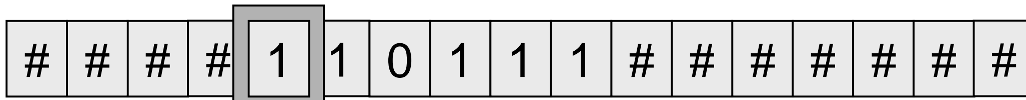
δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

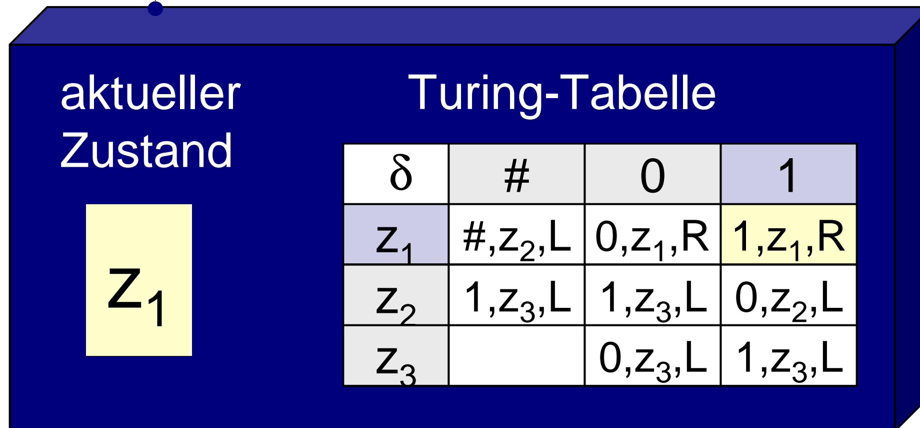
- Hardware:
  - Alphabet Σ
  - Unbegrenztes Band für Zeichen aus Σ
  - Beweglicher Lese-Schreibkopf
  - Zustände Q

- Software: Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Aus
  - aktuellem Zustand q
  - aktuellem Zeichen a
- gehe in
  - neuen Zustand
  - schreibe neues Zeichen
  - Bewege Kopf nach rechts (R) oder links (L)
- fertig,
  - falls Haltezustand erreicht
  - oder kein Eintrag in Tafel

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



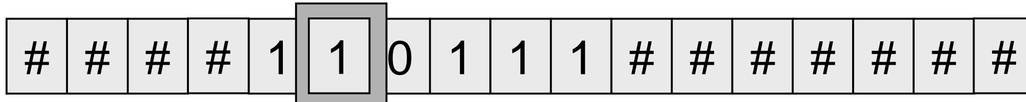


δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_1

Turing-Tabelle

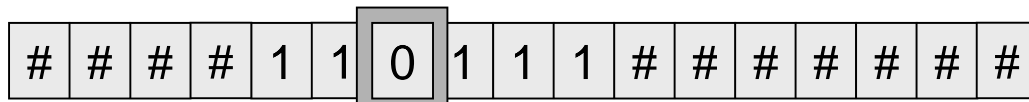
δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

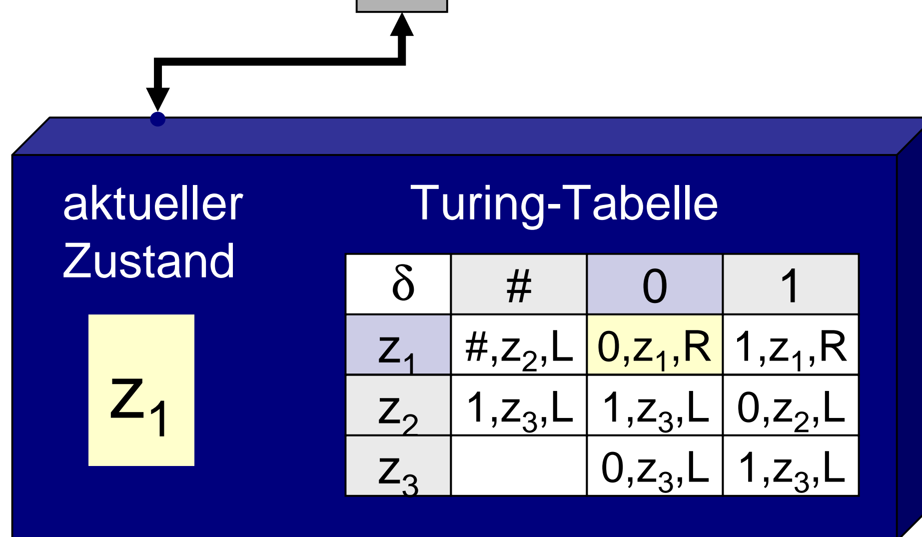
- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$

- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



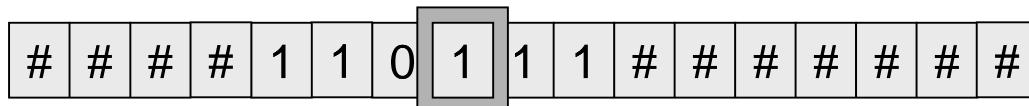


δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_1

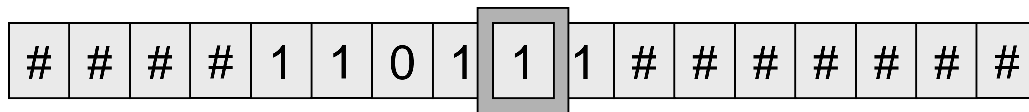
Turing-Tabelle

δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_1

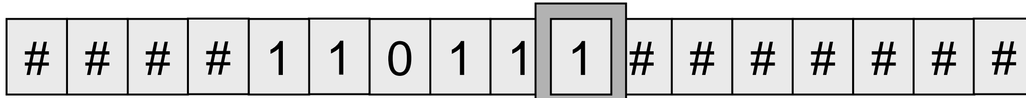
Turing-Tabelle

δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_1

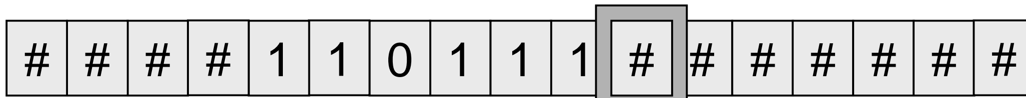
Turing-Tabelle

δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_1

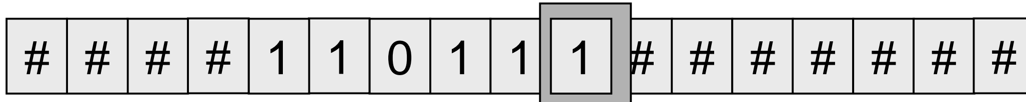
Turing-Tabelle

δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_2

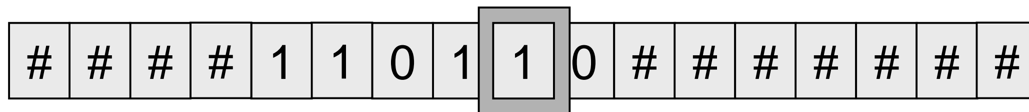
Turing-Tabelle

δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_2

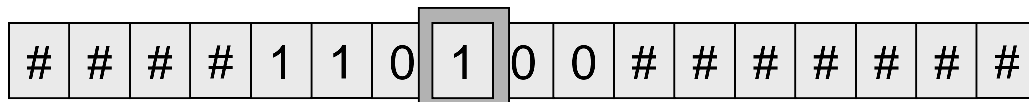
Turing-Tabelle

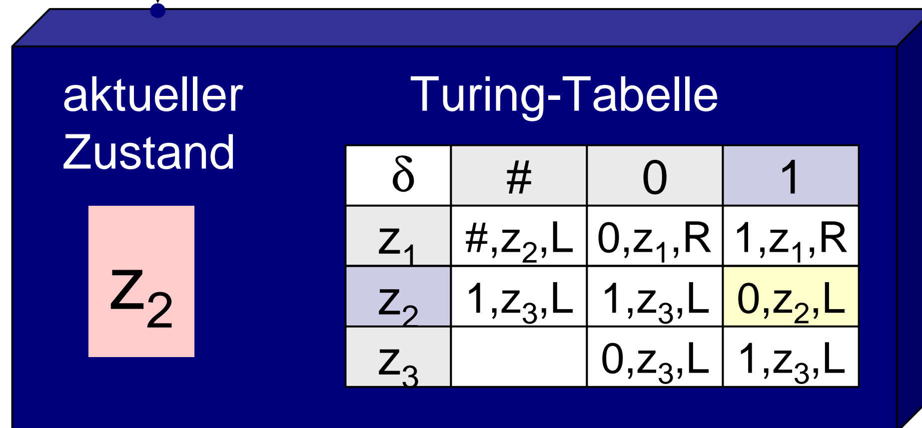
δ	#	0	1
z_1	#,z_2,L	0,z_1,R	1,z_1,R
z_2	1,z_3,L	1,z_3,L	0,z_2,L
z_3		0,z_3,L	1,z_3,L

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



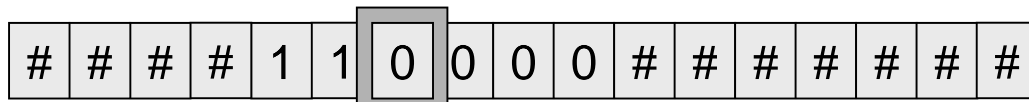


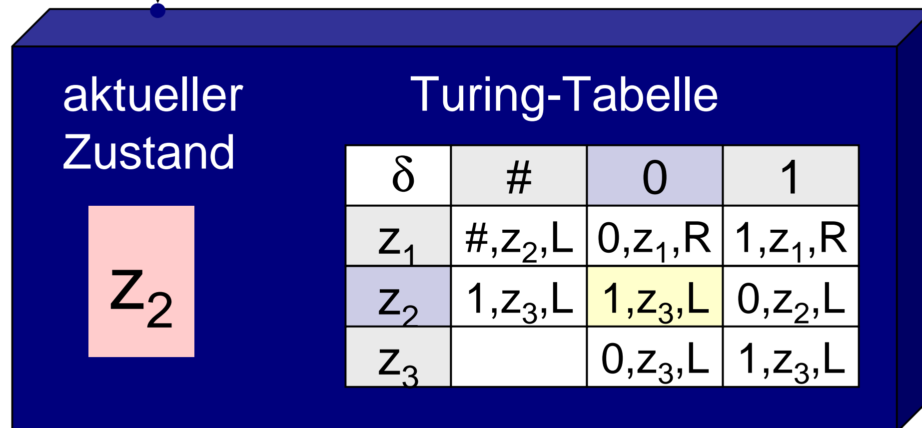
δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“





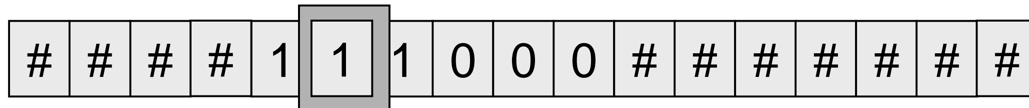
δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

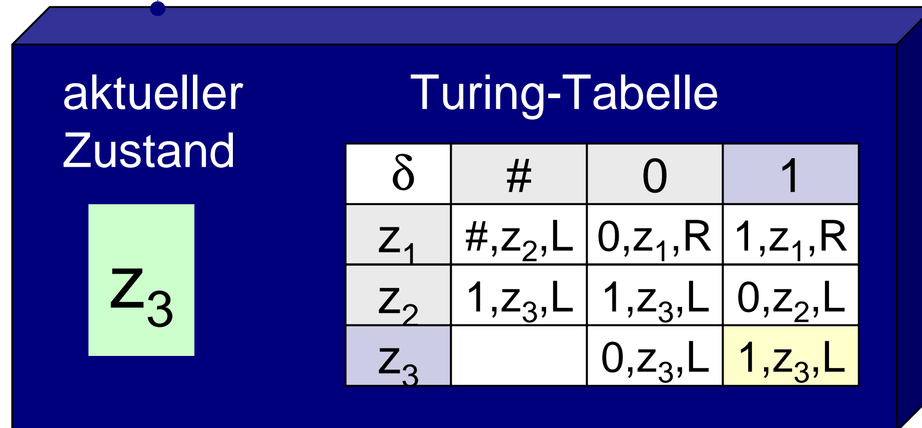
- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$

- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



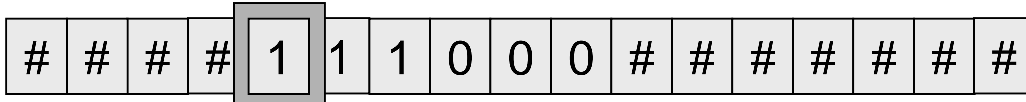


δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“



aktueller Zustand

z_3

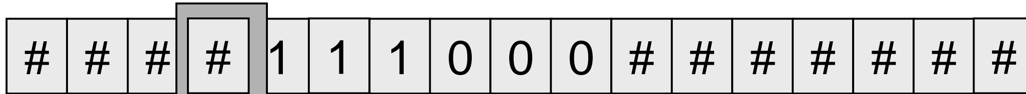
Turing-Tabelle

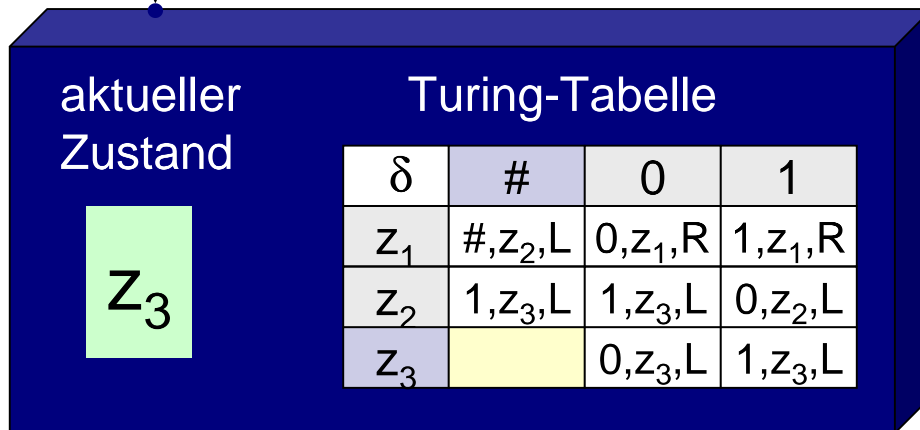
δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine „binary add 1“





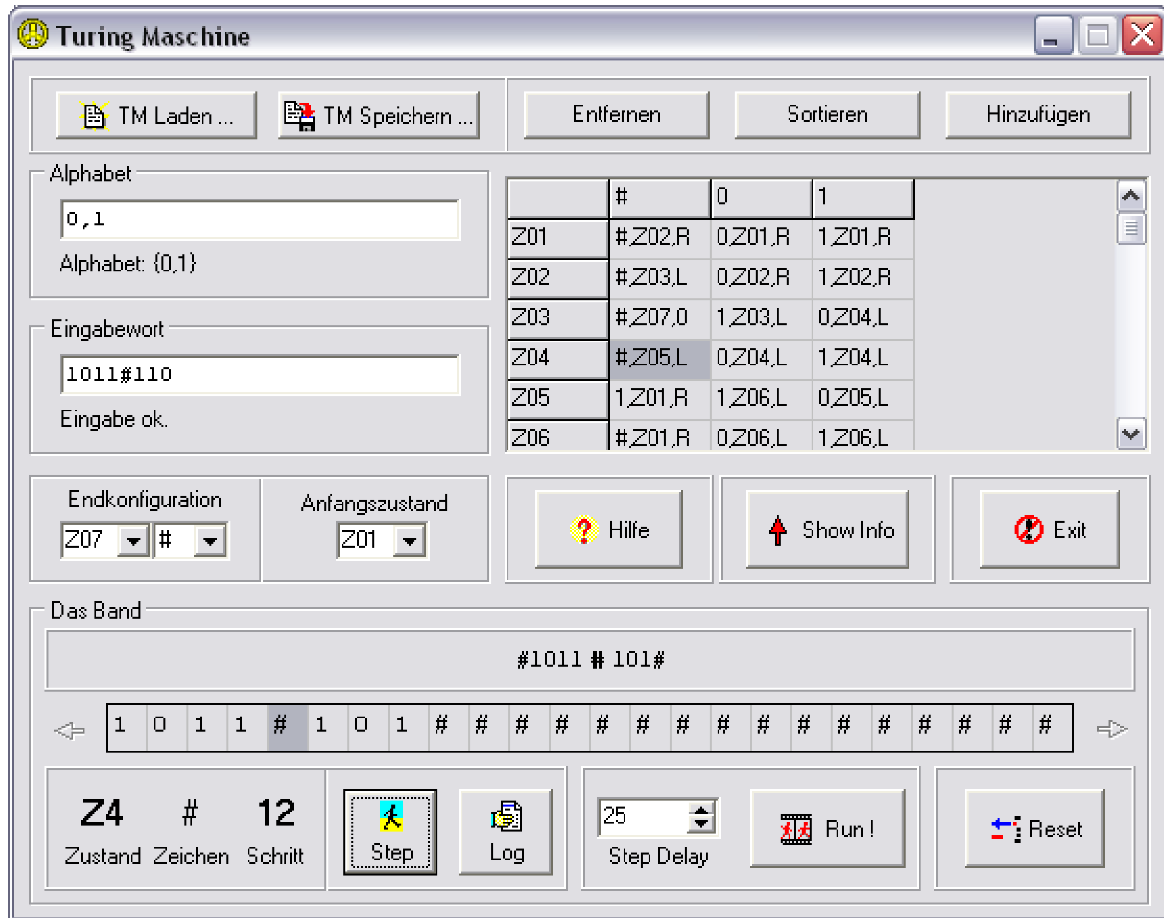
δ	#	0	1
z_1	$\#,z_2,L$	$0,z_1,R$	$1,z_1,R$
z_2	$1,z_3,L$	$1,z_3,L$	$0,z_2,L$
z_3		$0,z_3,L$	$1,z_3,L$

- Turingmaschine
  - Alphabet 0,1,# (#=leeres Kästchen)
  - Zustände $Q=\{z_1,z_2,z_3\}$
- Turing-Tabelle
  - $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$
- Addiere 1 zu der Binärzahl auf dem Band
- Idee
  - z_1: Gehe nach rechts
  - z_2: Addiere 1
  - z_3: Zurück an Anfang

© H. Peter Gumm, Philipps-Universität Marburg

Simulatoren

- Es sind eine Reihe von guten TM-Simulatoren im Internet verfügbar.
  - Nett zum Spielen.
  - Gut um Gefühl zu bekommen, dass man im Prinzip alles berechnen kann und
  - wie aufwendig es werden kann.

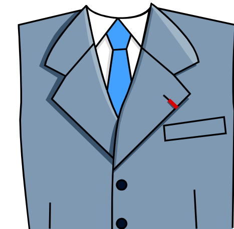


Turing-Maschinen-Simulator von Gregor Buchholz

© H. Peter Gumm, Philipps-Universität Marburg

Turingmaschine formale Definition

- $T=(Q, \Gamma,\delta,q_0,E)$ wobei



  - Q endliche Menge (Zustände)
  - Γ endliche Menge, $\# \in \Gamma$ (Alphabet)
  - $q_0 \in Q$ (Anfangszustand)
  - $E \subseteq Q$ (Endzustände)
  - $\delta: Q\times \Gamma \to Q\times \Gamma \times\{L,R\}$ (Turingtabelle)

- Konfiguration einer Turingmaschine:
  - $\alpha\ q\ \beta$ mit $\alpha,\beta \in \Gamma^*$

    Intuitiv: α, β : Bandinhalt links, bzw. rechts vom Kopf, akt. Zustand q.
    Alles außerhalb von $\alpha\beta$ ist #.

- Übergangsrelation $\alpha\ q\ \beta \vdash \alpha'\ q\ \beta'$ wobei $\alpha = a_1 \ldots a_m$ und $\beta = b_1 \ldots b_n$

  - $$a_1\ldots a_m\ q\ b_1 b_2 \ldots b_n \vdash \begin{cases} a_1\ldots a_m\ c\ q'\ b_2\ldots b_n & \text{falls } \delta(q,b_1) = (q',c,R) \\ a_1\ldots a_{m-1}\ q'\ a_m c\ b_2\ldots b_n & \text{falls } \delta(q,b_1) = (q',c,L) \end{cases}$$

  - Falls $\alpha=\varepsilon$ oder $\beta=\varepsilon$ ersetze sie vorher durch #

- $\vdash^*$ sei die reflexiv-transitive Hülle von $\vdash$

© H. Peter Gumm, Philipps-Universität Marburg

Turing-berechenbar

- Sei $\Sigma = \Gamma$-{#}. Eine partielle Funktion $f: \Sigma^* \to \Sigma^*$ heißt Turing-berechenbar, wenn es eine Turingmaschine gibt, mit

  $f(u)=w \Leftrightarrow q_0 u \mid\!-^* q_e w\#$ mit $q_e \in E$

  für alle $u,w \in \Sigma^*$.

- Eine Funktion $f: N^k \to N$ heißt Turing-berechenbar, falls es eine TM gibt mit

  $f(n_1,\ldots,n_k) = m \Leftrightarrow \#q_0 bin(n_1)\#\ldots\#bin(n_k)\# \mid\!-^* q_e bin(m)\#$ mit $q_e \in E$

- Beispiel: Wir haben bereits gesehen:

  $f(n)=n+1, \quad g(m,n) = m+n$

  sind Turing-berechenbar

© H. Peter Gumm, Philipps-Universität Marburg

Varianten

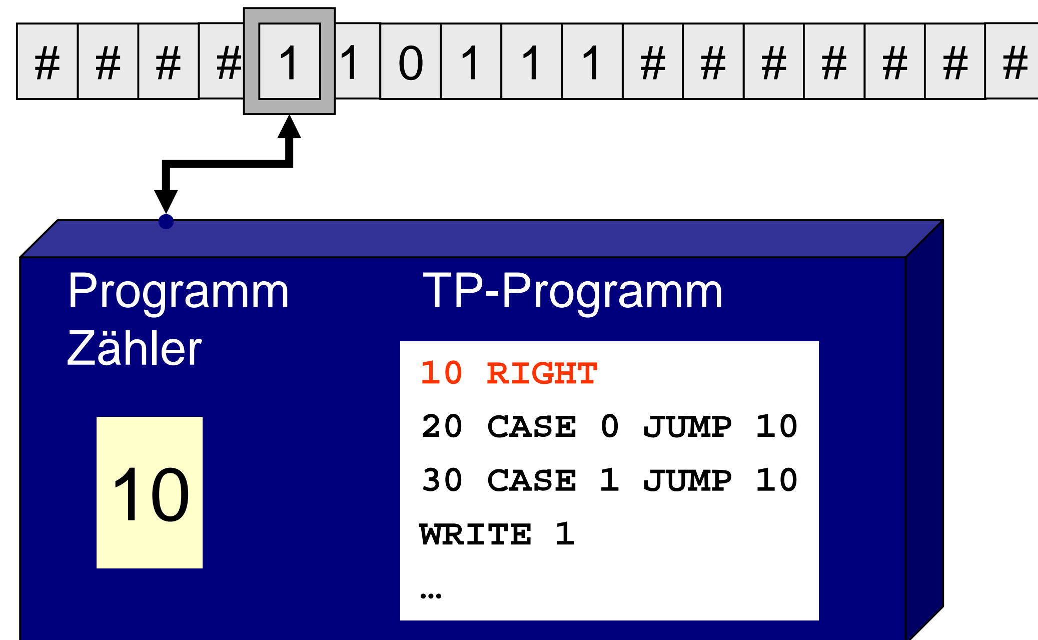
- Es gibt viele Variationen von Turing-Maschinen.
  - Sie sind jedesmal äquivalent, weil man je eine durch die andere simulieren kann
- Beispiele:
  - Erlaube, dass der Kopf sich nicht bewegt: $\delta(q,a)=(r,b,-)$
    - Ersetze $\delta(q,a)=(r,b,-)$ durch
      - $\delta(q,a)=(q',b,R)$ mit neuem Zustand q' und
      - $\delta(q',x)=(r,x,L)$ für jedes $x\in\Gamma$.
  - Einseitig unendliches Band (Idee):
    - Markiere das Ende durch Sonderzeichen
    - Betrachte gerade Positionen als rechtes Bandteil und ungerade als linkes Bandteil
    - ein Zustand hält fest, in welchem Bandteil sich der Kopf befindet,
    - Bewegung nach links/rechts im rechten Teil $\approx$ zwei Schritte nach links/rechts
    - Bewegung nach links/rechts im linken Teil $\approx$ zwei Schritte nach rechts/links
    - Erreichen des Sonderzeichens: Wechsel vom linken in rechten Teil oder umgekehrt
  - mehrere Bänder
    - siehe Literatur

© H. Peter Gumm, Philipps-Universität Marburg

Turing-Post Programme

Emil Leon Post
1897-1954

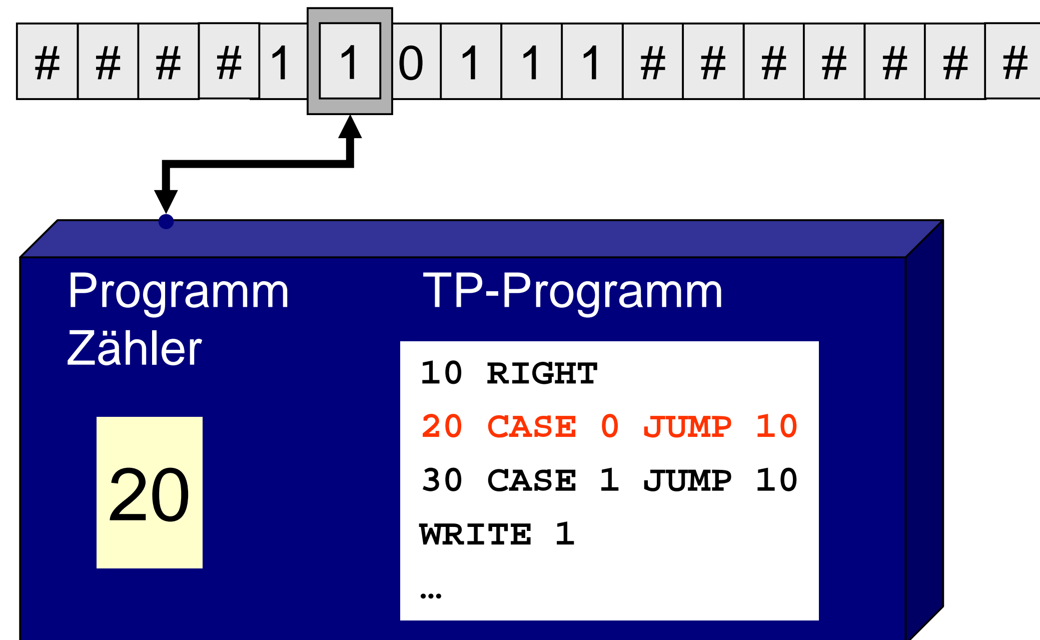
- Programmiersprache für TM
  - ersetzt Turingtabellen
- Programme bestehen aus Zeilennummern und Befehlen
  - analog Basic
- Mögliche Befehle:
  - **Left**
  - **Right**
  - **Write <e>**
  - **CASE <e> JUMP <n>**
  - **HALT**



Turing-Post Programme

Emil Leon Post
1897-1954

- Programmiersprache für TM
  - ersetzt Turingtabellen
- Programme bestehen aus Zeilennummern und Befehlen
  - analog Basic
- Mögliche Befehle:
  - **Left**
  - **Right**
  - **Write <e>**
  - **CASE <e> JUMP <n>**
  - **HALT**

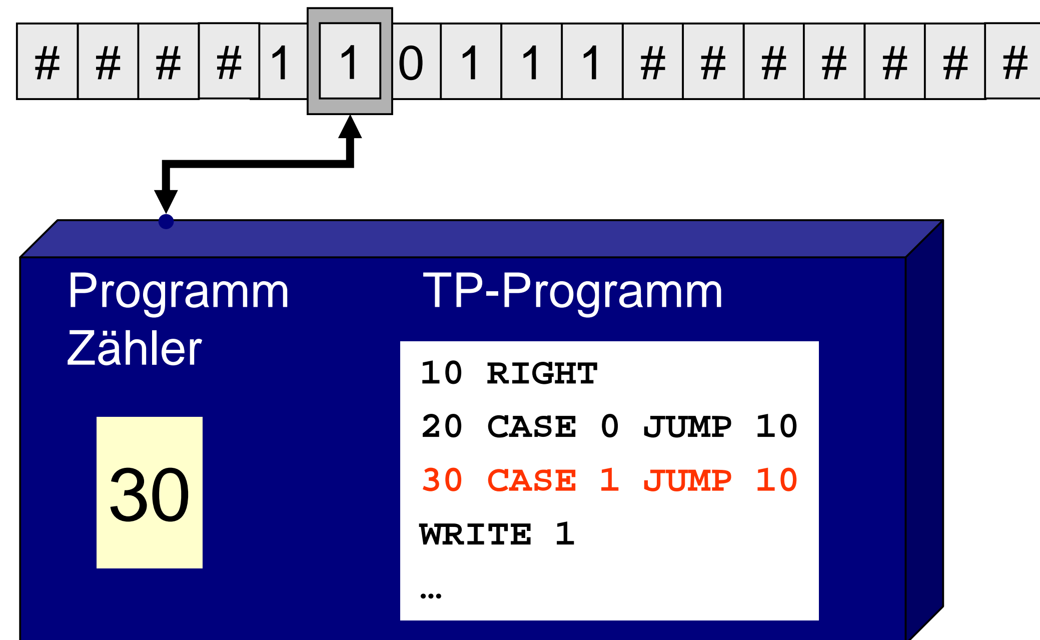


© H. Peter Gumm, Philipps-Universität Marburg

Turing-Post Programme

Emil Leon Post
1897-1954

- Programmiersprache für TM
  - ersetzt Turingtabellen
- Programme bestehen aus Zeilennummern und Befehlen
  - analog Basic
- Mögliche Befehle:
  - **Left**
  - **Right**
  - **Write <e>**
  - **CASE <e> JUMP <n>**
  - **HALT**

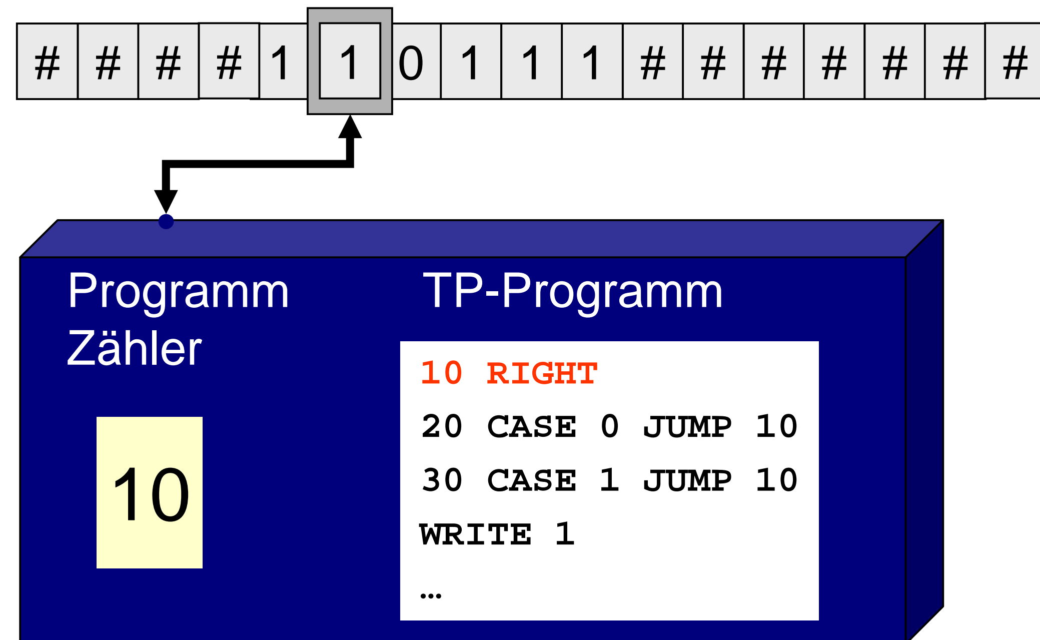


© H. Peter Gumm, Philipps-Universität Marburg

Turing-Post Programme

Emil Leon Post
1897-1954

- Programmiersprache für TM
  - ersetzt Turingtabellen
- Programme bestehen aus Zeilennummern und Befehlen
  - analog Basic
- Mögliche Befehle:
  - **Left**
  - **Right**
  - **Write <e>**
  - **CASE <e> JUMP <n>**
  - **HALT**

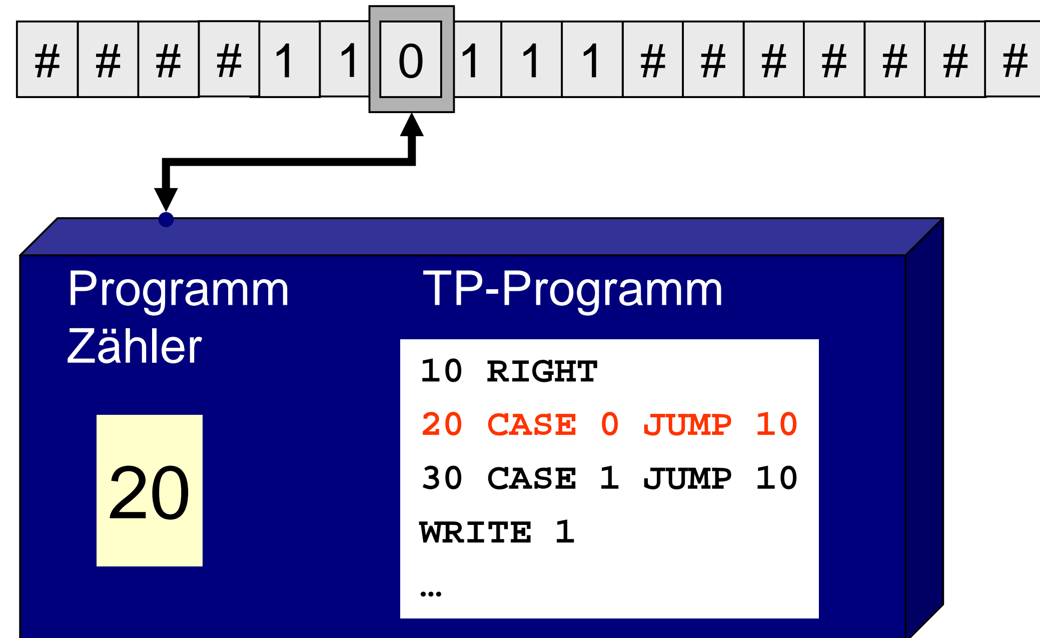


© H. Peter Gumm, Philipps-Universität Marburg

Turing-Post Programme

Emil Leon Post
1897-1954

- Programmiersprache für TM
  - ersetzt Turingtabellen
- Programme bestehen aus Zeilennummern und Befehlen
  - analog Basic
- Mögliche Befehle:
  - **Left**
  - **Right**
  - **Write <e>**
  - **CASE <e> JUMP <n>**
  - **HALT**



Beispiel: TP-Programm für „+1“

```
// Über die erste Zahl hinweg nach rechts
10 RIGHT
20 CASE 0 JUMP 10
30 CASE 1 JUMP 10
// Von rechts kommend 1-en zu 0-en umwandeln
40 LEFT
50 CASE 1 JUMP 80
60 CASE 0 JUMP 100
70 CASE # JUMP 100
80 WRITE 0
90 CASE 0 JUMP 40
100 WRITE 1
// Nach links laufen
110 LEFT
120 CASE 0 JUMP 110
130 CASE 1 JUMP 110
140 HALT
```

- Zeilennummern übernehmen Rolle der Zustände
- Lesen, Schreiben und Bewegung explizit und voneinander unabhängig
- Original-Zustände anhand der Kommentare noch erkennbar

© H. Peter Gumm, Philipps-Universität Marburg

Erweiterungen

- Neue TP-Befehle lassen sich als Makros definieren

Falls $\Sigma = \{a_1,\ldots,a_n\}$:

```
JUMP <n> :=
    CASE a1 JUMP <n>
          …
    CASE an JUMP <n>
```

- Statt Zeilennummern kann man symbolische *label* verwenden
  - Sind leicht in Zeilennummern zu verwandeln

```
rechts:   RIGHT
           CASE # JUMP links
           JUMP rechts
links:    LEFT
           CASE 0 JUMP eins
           CASE # JUMP eins
null:     WRITE 0
           JUMP links
eins:     WRITE 1
done:     LEFT
          CASE # JUMP ende
          JUMP done
ende:     HALT
```

© H. Peter Gumm, Philipps-Universität Marburg

Turingprogramm in Turingtafel

- Setze Anweisungen um in Einträge in Turingtabelle
  - Programmzeile n à Zustand z_n

```
n :     Write e
```

$\delta(z_n,x) = (z_{n+1}, e, -)$ für alle $x \in \Sigma$

```
n :     Left
```

$\delta(z_n,x) = (z_{n+1}, x, L)$ für alle $x \in \Sigma$

```
n : CASE e JUMP m
```

$$\delta(z_n,x) = \begin{cases} (z_m, e, -) & \text{falls } x=e \\ (z_{n+1}, x, -) & \text{sonst} \end{cases}$$

Turingtafel zu TP-Programm

- Führe labels ein:
  - [i] für jedes $q_i \in Q$
  - [i,k] für jedes $q_i \in Q$ und $a_k \in \Sigma$
- Der Eintrag $\delta(q_i, a_k) = (q_j, b, L)$ wird zum Programmtext:

```
// Code für Zustand qi
[i]:    CASE a1 JUMP [i,1]
         …
        CASE ak JUMP [i,k]
        …
// Code für Zustand qi+1
[i+1]: …
```

```
// Code für Zustand delta(qi,ak)
[i,k]: Write b
       Left  //analog Right
       JUMP [j]      …
```

© H. Peter Gumm, Philipps-Universität Marburg

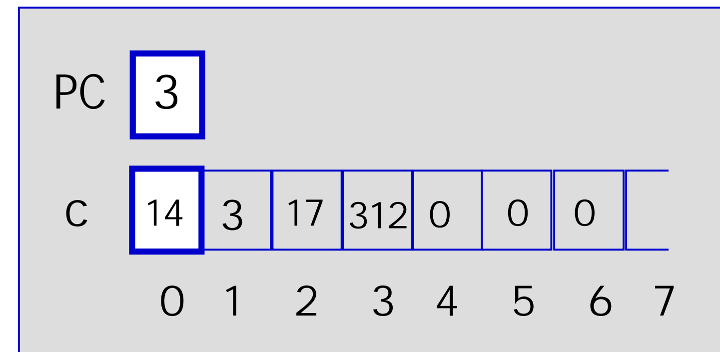
Nachteil von Turingmaschinen

- Turingmaschine kann Zwischenergebnisse auf dem Band zwischenspeichern und wieder lesen
  - viele Kopfbewegungen notwendig
  - umständlich zu programmieren
  - sehr aufwendig
- Erleichterung
  - TM mit mehreren Bändern
  - immer noch weit weg von realen Maschinen
- Realistischeres Modell:
  - Registermaschinen
  - direkter R/W-Zugriff

© H. Peter Gumm, Philipps-Universität Marburg

Registermaschinen

- Hardware:
  - Unbegrenztes Array von Speicherzellen c[1], c[2], ...
    - Jede kann eine nat. Zahl speichern
  - Akkumulator c[0]
  - Programmzähler PC



- Befehle
  - Nummerierte Programmzeilen
  - Load n — c[0] := n
  - Load [n] — c[0] := c[n]
  - Store [n] — c[n] := c[0]
  - Add [n] — c[0] := c[0]+c[n]
    - Sub^+, Mul, Div — analog
  - Goto i — PC := i
  - JZero i — PC := i falls c[0]=0.

Sub^+ [n] sei definiert als c[0] = max(0,c[0]-c[n])

© H. Peter Gumm, Philipps-Universität Marburg

RM-Berechenbarkeit

- Um eine Funktion $f: N^k \to N^r$ mit einer RM zu berechnen :
  - Schreibe Argumente $x_1,\ldots,x_k$ in Zellen c[1],...,c[k].
  - Alle anderen Zellen enthalten 0
  - Wenn Programm terminiert, finde Ergebnis in c[1], ...,c[r]
- Eine partielle Funktion $f :: N^k \to N^r$ heißt RM-berechenbar, falls es ein RM-Programm gibt, das f im obigen Sinn berechnet.
  - $f(x_1,..,x_k) = \bot$

    $\Leftrightarrow$ Berechnung terminiert nicht

ggT

```
0:   Load [1]
1:   Sub+ [2]
2:   JZero 5
       // hier: c[1] > c[2]
3:   Store [1]
4:   Goto 0
       // hier: c[1] <= c[2]
5:   Load [2]
6:   Sub+ [1]
7:   JZero 11
       // hier: c[2] > c[1]
8:   Store [2]
9:   Goto 0
       // hier: c[1]=c[2]
11: Halt
```

© H. Peter Gumm, Philipps-Universität Marburg

Zusätzliche Operationen

- Zusätzliche Operationen lassen sich leicht als Makros implementieren. Z.B.:
  - JNZ i
  - CMP [n]
  - JEq i
  - JEq i
  - JLess i
  - JEq i

JNZ // Vorsicht: überschreibt c[k]

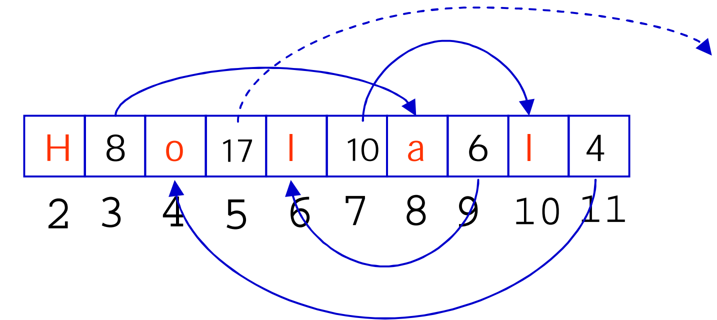
```
      Store [k]         // Akku speichern
      Load  1           //
      Sub+  [k]         // 1- Akku
      JZero  springe    // Sprung, wenn alter Akku > 0
      Load [k]
      Goto done
   springe:
      Load [k]
      Goto i
   done :
```

- Variablennamen statt Zellennummern
  - z.B. „summe“ statt c[17]

© H. Peter Gumm, Philipps-Universität Marburg

Mächtigkeit

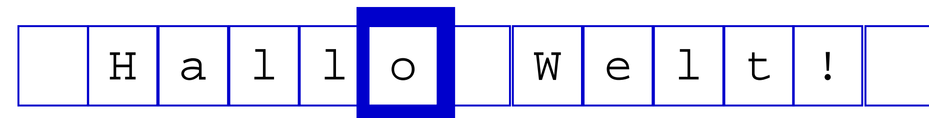
- Registermaschinen können Turingmaschinen simulieren



  - Überlegen Sie sich eine Implementierung verketteter Listen:

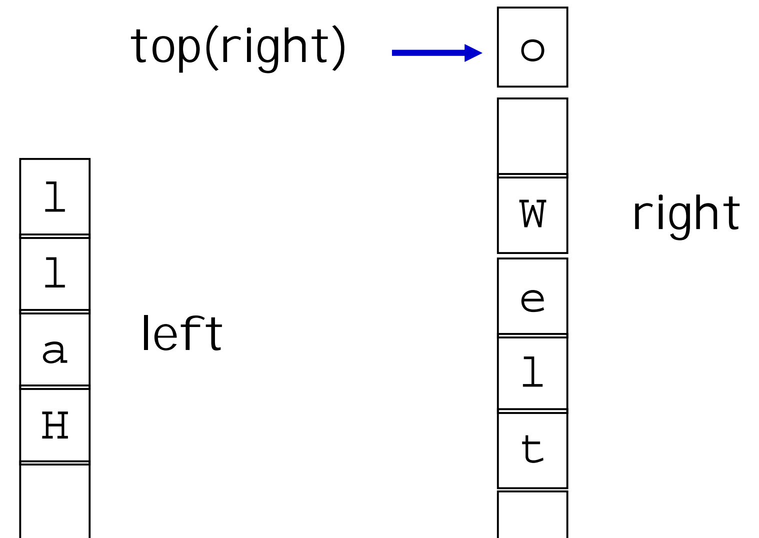
    Zeiger auf Anfang

    Zellen aus zwei Zahlen: inhalt, next



  - Repräsentiere
    - Zustände, Zeichen, als Zahlen
    - c[1] = Programmzähler
    - Bandinhalt durch zwei Listen
      - left = links vom Kopf
      - right = rechts vom Kopf

  - Befehle
    - Right : pop[right]; push[left]
    - Left : pop[left]; push[right]
    - Write e : pop[right]; Load e; push[right]
    - Case e Jump n: Load e; CMP top(right); JEQ n



© H. Peter Gumm, Philipps-Universität Marburg

Mächtigkeit

- Turingmaschinen können Registermaschinen simulieren
  - Am besten zuerst Mehrbandturingmaschine simulieren
  - Mehrband-TM verwenden:
    - Ein Band für Speicherinhalte
    - Ein Band für Akku
    - Ein Band für Programmzähler
  - Mühsam aber langweilig
    - Glauben Sie es oder programmieren Sie es
- Turingmaschinen und Registermaschinen gleichmächtig
- Partielle Funktion $f :: N^k \rightarrow N$ ist Turing-berechenbar $\Leftrightarrow$ f ist RM-berechenbar

© H. Peter Gumm, Philipps-Universität Marburg

Inhalt

1. Turingmaschinen
   - Definition und Beispiele
   - Turing-Post Programme
   - Registermaschinen
   - Äquivalenz

2. Programmierparadigmen
   - GOTO
   - LOOP- und WHILE-Sprachen
   - Äquivalenz von GOTO und WHILE
   - Churchsche These

3. Rekursive Funktionen
   - Primitive Rekursion
   - Äquivalenz zu LOOP
   - μ-Rekursion
   - Äquivalenz zu While

4. Halteproblem
   - Aufzählbarkeit u. Entscheidbarkeit
   - Halteproblem
   - Nichtberechenbare Funktionen
   - Satz von Rice

© H. Peter Gumm, Philipps-Universität Marburg

GOTO-Programme

- Ähnlich wie RM, nur
  - Variablennamen statt Speicherzellen
  - Einfache arithmetische Ausdrücke
    - *Expr*
  - Vergleichsausdrücke
    - *Bexpr*
  - Anweisungen
    - Zuweisung
    - Bedingter Sprung
  - Datentyp: Nat

Goto

```
Stmt   :: <n> : id := Expr
        | <n> : if Bexpr Goto <m>

Expr   :: id | 0 | succ(Expr)

Bexpr  :: Expr <= Expr
```

Semantik: offensichtlich

<n>, <m> : Zeilennummern

© H. Peter Gumm, Philipps-Universität Marburg

Goto-berechenbar

- Berechnung einer Funktion $f:N^k \to N^r$:
  - Argumente $(n_1,..,n_k)$ in Variablen $x_1,..,x_k$.
  - Starte Programm
  - Wenn es terminiert erwarte Ergebnis in Variable $y_1,...,y_r$.
- Eine partielle Funktion $f :: N^k \to N^r$ heißt Goto-berechenbar, wenn es ein Goto-Programm gibt, das
  - terminiert gdw. $f(n_1,..,n_k) \neq \bot$
  - dabei f in dem obigen Sinne berechnet

© H. Peter Gumm, Philipps-Universität Marburg

Goto-Programm kann RM simulieren

- Relativ einfach
  - Zunächst primitive Operationen implementieren, + für Add, - für Sub⁺, etc.

Makro für z := x+y

```
10 : z := x
20 : k := 0
30 : if y <= k goto 70
40 : z := succ(z)
50 : k := succ(k)
60 : if k <= y goto 40
70 :
```

- Führe Variablen für alle benötigten Zellen ein
  - akku, c1, c2, ... ,
- Danach Übersetzung fast 1-1

© H. Peter Gumm, Philipps-Universität Marburg

RM kann Goto-Programm simulieren

- Ordne jeder Variablen y eine Speicherzelle c_y zu

- Makro für Zuweisung

RM-Makro für z := x+y

```
Load  [c_x]
Add   [c_y]
Store [c_z]
```

- Makro für
  - if x <= y Goto <n>

RM-Makro für if x <= y Goto <n>

```
Load [c_x]
CMP  [c_y]
JLE <n>
```

© H. Peter Gumm, Philipps-Universität Marburg

While-Programme

- Minimalistische Sprache
  - Prototypisch
- Keine Zeilennummern
  - Keine Sprünge
- Semikolon „;“ bedeutet Hintereinander-ausführung
- Klammern optional
  - begin … end
  - { … }
- Stattdessen: Einrückung

While

```
Stmt    :: id := Expr
         | Stmt ; Stmt
         | if Bexpr then Stmt else Stmt
         | while Bexpr do Stmt

Expr    :: id | 0 | succ(Expr)

Bexpr   :: Expr <= Expr
```

© H. Peter Gumm, Philipps-Universität Marburg

While-berechenbar

- Analog zu Goto-berechenbar:
  - Argumente $(n_1,..,n_k)$ in Variablen $x_1,..,x_k$.
  - Starte Programm
  - Wenn es terminiert erwarte Ergebnis in Variablen $y_1,...,y_r$.
- Eine partielle Funktion $f :: N^k \to N^r$ heißt While-berechenbar, wenn es ein While-Programm gibt, das
  - terminiert gdw. $f(n_1,..,n_k) \neq \bot$
  - dabei f in dem obigen Sinne berechnet

© H. Peter Gumm, Philipps-Universität Marburg

Goto simuliert While

- Jede While-Anweisung kann durch ein Goto-Programmstück simuliert werden
  - rekursiv

While E_1 <= E_2 do
   $Stmt$

(While)

n$_1$: if succ(E_2) <= E_1 Goto n$_2$
   Übersetzung($Stmt$)
   Goto n$_1$
n$_2$:

if E_1 <= E_2
   then $Stmt_1$
   else $Stmt_2$

(if-then-else)

n$_1$: if E$_1$ <= E$_2$ Goto n$_2$
   Übersetzung(Stmt$_2$)
   Goto n$_3$
n$_2$: Übersetzung(Stmt$_1$)
n$_3$:

© H. Peter Gumm, Philipps-Universität Marburg

While simuliert Goto

- Jedes Goto-Programm kann durch ein While-Programm simuliert werden
- Idee:
  - Führe Programmzähler ein: PC
  - Übersetze jede Zeile G_i des Goto-Programms in eine Anweisung W_i des While-Programms

G_i	W_i
<n> : id := *Expr*	Übersetzung: id := *Expr* ; PC := succ(PC)
<n> : if *Bexpr* Goto *<k>*	Übersetzung: if *Bexpr* then PC := k else PC := succ(PC)

© H. Peter Gumm, Philipps-Universität Marburg

While simuliert Goto

Stephen Cole Kleene
1909 - 1994

- Jedes Goto-Programm kann durch ein While-Programm simuliert werden
- Man kommt mit einem einzigen While aus

Goto-Programm

```
1 : G1
2 : G2
 ...
k : Gk
```

While-Programm

```
PC := 1;
While PC <= k do {
   if PC <= 1 then W1  else
     if PC <= 2 then W2  else
                 ...
       if PC <= k then Wk
}
```

- While-Programme und Goto-Programme sind äquivalent
- (Kleene) : Eine While-Schleife genügt – auch für While-Programme.

© H. Peter Gumm, Philipps-Universität Marburg

Äquivalenzsatz

- Für eine partielle Funktion $f : N^k \to N$ sind äquivalent:
  - f ist Turing-berechenbar
  - f mit TP-Programm berechenbar
  - f ist RM-berechenbar
  - f ist Goto-berechenbar
  - f ist While-berechenbar
- Dies ist eine erste Bestätigung der Church'schen These
- Von jetzt an können wir z.B. gleichsetzen:
  - Algorithmus = While-Programm

© H. Peter Gumm, Philipps-Universität Marburg

Algorithmenbegriffe

- Es gibt viele Vorschläge für eine saubere mathematische Definition von Algorithmus und zugehörige partielle berechnete Funktion

- Ein Algorithmus ist

  - eine Turingmaschine
    - eine Register-, 2-Stack-Maschine

  - ein While-Programm
    - ein Smalltalk-, Java-, Pascal-, Basic-Programm

  - eine rekursiv definierte Funktion
    - ein Haskell-, ML-, LISP-Programm

  - eine logisch definierte Funktion
    - ein Prolog-Programm

- Erstaunlicherweise stellte sich immer wieder heraus :

  - alle diese Definitionen sind äquivalent
  - die berechenbaren Funktionen sind immer die gleichen

© H. Peter Gumm, Philipps-Universität Marburg

Church's These

Alonzo Church

1903-1995

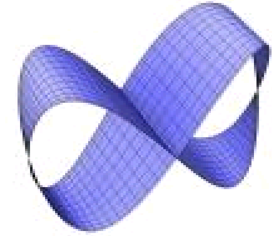
- Jede vernünftige Definition von Algorithmus und berechenbare Funktion führt auf die gleiche Klasse von Funktionen –

  die partiell berechenbaren Funktionen

  - Scheint zu stimmen
  - Immer wieder bestätigt
  - Kann man nicht beweisen
  - Sehr praktisch
    - Jeder kann sich seine Lieblingsdefinition aussuchen
    - Z.B. Java, Prolog, Haskell,...

© H. Peter Gumm, Philipps-Universität Marburg

Die Sprache LOOP



- Idee: Ersetze While-Schleife durch FOR-Schleife
- Syntax: … ansonsten wie WHILE

LOOP

```
Stmt     :: id := Expr
          |  Stmt ;  Stmt
          | if Bexpr then Stmt else Stmt
          | for x := Expr to Expr do Stmt

Expr     :: id | 0 | succ(Expr)

Bexpr    :: Expr <= Expr
```

© H. Peter Gumm, Philipps-Universität Marburg

Semantik von FOR

```
for x := Expr1 to Expr2 do Stmt
```

- Werte $Expr_1$ und $Expr_2$ aus
  - Ergebnisse seien n_1 und n_2
- Falls $n_1 > n_2$ ist die Anweisung beendet
- Falls $n_1 \leq n_2$ durchläuft x die Zahlen von n_1 bis n_2 und jedesmal wird Stmt ausgewertet
- Innerhalb Stmt
  - darf x gelesen werden
  - darf x nicht verändert werden

- Konsequenz:
  - Jedes LOOP-Programm terminiert
  - Jede LOOP-berechenbare Funktion ist total

© H. Peter Gumm, Philipps-Universität Marburg

Jedes Polynom ist LOOP-berechenbar

- Addition
  - ```
    z:= x1;
    for x := 1 to x2 do
        z := succ(z);
    ```
    ($z := x_1$; for $x := 1$ to x_2 do $z := succ(z)$;)
  - Wir können ab jetzt so tun, als ob z := x+y erlaubt wäre
- Multiplikation
  - ```
    z := 0;
    for x := 1 to x2 do
      z := z+x1
    ```
    ($z := 0$; for $x := 1$ to x_2 do $z := z+x_1$)
- Exponentiation
  - ```
    z := x1;
    for x := 1 to x2 do
      z := z*x1
    ```
    ($z := x_1$; for $x := 1$ to x_2 do $z := z*x_1$)

© H. Peter Gumm, Philipps-Universität Marburg
    ```

Effiziente Algorithmen sind LOOP-berechenbar

- Jeder Algorithmus polynomialer Komplexität $O(n^k)$ ist LOOP-berechenbar

- <u>Beweis</u> (Der Einfachheit halber nur eine Input-Variable x):
  Es gibt ein c so dass A(x) höchstens $c*x^k$ viele Schritte benötigt.

- Algorithmus lässt sich schreiben (wg. Kleene) als

  S_1 ; While Bexpr do S_2

  wobei in S_1 und in S_2 kein While mehr vorkommt. S_1 und in S_2 sind also schon LOOP-Programme.

  Das Ganze ist äquivalent zu folgendem Algorithmus:

```
maxstep := c*x^k ;                    // LOOP-berechenbar
S1 ;
FOR step := 1 TO maxstep DO
    IF Bexpr then S2
```

  wobei step eine Variable ist, die nicht in S_2 vorkommt.

© H. Peter Gumm, Philipps-Universität Marburg

Inhalt

1. Turingmaschinen
   - Definition und Beispiele
   - Turing-Post Programme
   - Registermaschinen
   - Äquivalenz

2. Programmierparadigmen
   - GOTO
   - LOOP- und WHILE-Sprachen
   - Äquivalenz von GOTO und WHILE
   - Churchsche These

3. Rekursive Funktionen
   - Primitive Rekursion
   - Äquivalenz zu LOOP
   - μ-Rekursion
   - Äquivalenz zu While

4. Halteproblem
   - Aufzählbarkeit u. Entscheidbarkeit
   - Halteproblem
   - Nichtberechenbare Funktionen
   - Satz von Rice

© H. Peter Gumm, Philipps-Universität Marburg

Rekursive Funktionen

- Funktionale Sprachen haben
  - keine Zuweisung
  - keine Schleifen
- stattdessen
  - Rekursion

- Vorteile
  - mathematische Beschreibung
  - weniger Fehler
  - rapid prototyping

- Nachteile
  - zu wenig mathematisch gebildete Programmierer
  - daher nicht mainstream
- aber
  - fast alle Sprachen erlauben zusätzlich Rekursion
  - moderne Sprachen integrieren funktionale Konzepte
    - Python, Ruby, Scala, C#

© H. Peter Gumm, Philipps-Universität Marburg

Primitiv rekursive Funktionen:

- Basisfunktionen
  - $f(x) = k$ — Konstante $k \in N$
  - $\pi^n_i(x_1,\dots,x_n) = x_i$ — i-te Projektion
  - $succ(x) = x+1$ — Nachfolger
- Abschluss unter
  - Komposition (Einsetzen)
  - primitives Rekursionsschema:

    Gegeben $g: N^k \to N$, $h: N^{k+2} \to N$

    - $f(0, x_1,\dots,x_k) := g(x_1,\dots,x_k)$
    - $f(n+1, x_1,\dots,x_k) := h(f(n, x_1,\dots,x_k), n, x_1,\dots,x_k)$

© H. Peter Gumm, Philipps-Universität Marburg

Beispiele

PR Funktionen

add(0,y) = y
add(x+1,y) = succ(add(x,y))

pred(0) = 0
pred(x+1) = x

sub(x,0) = x
sub(x,y+1) = pred(sub(x,y))

mult(x,0) = 0
mult(x,y+1) = add(mult(x,y),x)

not(0) = 1
not(x+1) = 0

Wir können 0 als false deuten und n+1 als true.

Hilfsfunkt. für PR-Schema

g(y) = π^1_1(y)
h(w,x,y) = succ(w)

g() = 0
h(w,x) = x

g(x) = x
h(x,y,w) = pred(w)

g(x) = 0
h(x,y,w) = add(w,x)

g() = 1
h(w,x) = 0

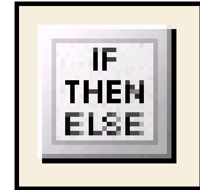
© H. Peter Gumm, Philipps-Universität Marburg

Argumentpositionen und Gleichheit

- Sei z.B. $g(x,y,z)$ primitiv rekursiv, dann auch z.B.:
  - $f_1(x,y) := g(x,x,y)$
  - $f_2(x,y,z) := g(y,z,x)$
  - $f_3(x,y,z,u) := g(x,y,z)$
- Komposition und Projektionen erlauben
  - Argumente gleichsetzen
    - $f_1(x,y) = g(\pi^2_1(x,y), \pi^2_1(x,y), \pi^2_2(x,y))$ p.r.
  - Argumente zu vertauschen
    - $f_2(x,y,z) = g(\pi^3_2(x,y,z), \pi^3_3(x,y,z), \pi^3_1(x,y,z))$ p.r.
  - Irrelevante Argumente hinzuzunehmen
    - $f_3(x,y,z,u) = g(\pi^4_1(x,y,z,u), \pi^4_2(x,y,z,u), \pi^4_3(x,y,z,u))$ p.r.
- Das führt nicht aus der Menge der p.r. Funktionen heraus

© H. Peter Gumm, Philipps-Universität Marburg

if-then-else ist p.r.



- Seien $B : N^k \rightarrow N$ sowie $f, g : N^k \rightarrow N$ p.r. , dann auch

$$ite(x_1,\ldots,x_k) := B(x_1,\ldots,x_k)\ ?\ f(x_1,\ldots,x_k)\ :\ g(x_1,\ldots,x_k)$$

Beweis:

- $ite(x_1,\ldots,x_k) := not(not(B(x_1,\ldots,x_k))) * f(x_1,\ldots,x_k) + not(B(x_1,\ldots,x_k)) * g(x_1,\ldots,x_k)$

  ist Komposition von p.r. Funktionen

© H. Peter Gumm, Philipps-Universität Marburg

p.r.-Funktionen sind LOOP-berechenbar

- Basisfunktionen
  - offensichtlich

Erinnerung:

$f(0, \quad x_1,\ldots,x_k) := g(x_1,\ldots,x_k)$

$f(n+1, x_1,\ldots,x_k) := h(f(n, x_1,\ldots,x_k), n, x_1,\ldots,x_k)$

- Komposition
  - Sind $g_1,..,g_k:N^r \to N$ und $h:N^k \to N$ LOOP-berechenbar, dann auch
    - $f(x_1,\ldots,x_r) := h(g_1(x_1,\ldots,x_r), \ldots, g_k(x_1,\ldots,x_r))$
  - Beweis:
    - Berechne nacheinander $g_1(x_1,\ldots,x_r), \ldots, g_k(x_1,\ldots,x_r)$ und speichere die Ergebnisse jeweils in $y_1,\ldots,y_k$.
    - Kopiere die $y_1,\ldots,y_k$ nach $x_1,\ldots,x_k$ und berechne h .

    Wenn die $g_1,..,g_k$ durch LOOP-Programme berechenbar sind, dann auch f.
- Primitives Rek.Schema
  - Sind g und h LOOP- berechenbar, dann auch $f : N^k \to N$ mit

```
w := g(x1,…,xk);
for z := 1 to n do
  w := h(w,z,x1,…,xk)
```

© H. Peter Gumm, Philipps-Universität Marburg

Mehrstellige Funktionen

- Eine Funktion $f : N^k \to N^r$ heißt p.r., falls alle Koordinatenfunktionen $f_1,\ldots,f_r$ p.r. sind
  - $f(x_1,\ldots,x_k) = (y_1,\ldots,y_r)$ gegeben durch
    - $f_1(x_1,\ldots,x_k) = y_1$
    - …
    - $f_r(x_1,\ldots,x_k) = y_r$

LOOP Programm

```
{
  x := 5;
  if n>0 then x:=x+n else n=n+1
}
```

$f(x,n)=(f_1(x,n),f_2(x,n))$ wobei

$$f_1(x,n) = if(>(n,0),5+n,5)$$
$$f_2(x,n) = if(>(n,0),n,n+1)$$

© H. Peter Gumm, Philipps-Universität Marburg

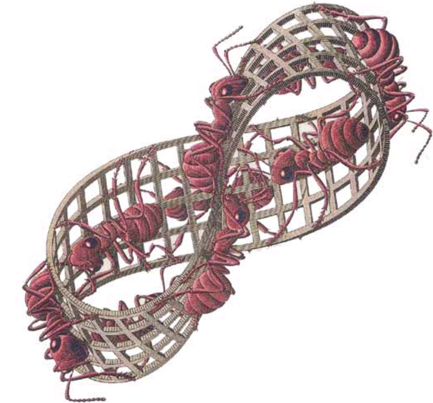
LOOP-Programme sind p.r.

- Umfassen $x_1,\dots,x_n$ alle Variablen in einem LOOP-Programm P, dann ist P durch eine n-stellige p.r.-Funktion f_P beschreibbar:
  - Zuweisung : $x_i := e(x_1,\dots,x_n)$
    - $f_{:=}(x_1,\dots,x_n) = (x_1,\dots,x_{i-1},e(x_1,\dots,x_n),x_{i+1}, \dots,x_n)$
  - Hintereinanderausführung: P;Q
    - $f_{P;Q}(x_1,\dots,x_n) = f_Q(f_P(x_1,\dots,x_n))$
  - Bedingung: if B then P else Q
    - $f_{if}(x_1,\dots,x_n)=ite(B(x_1,\dots,x_n),f_P(x_1,\dots,x_n)\ f_P(x_1,\dots,x_n))$
  - Iteration : FOR k:=0 TO $e(x_1,\dots,x_n)$ DO P
    - $f_{FOR}(x_1,\dots,x_n) = g(e(x_1,\dots,x_n),x_1,\dots,x_n)$ wobei

$$g(0,x_1,\dots,x_n) = (x_1,\dots,x_n)$$
$$g(k+1,x_1,\dots,x_n) = f_P(g(k,x_1,\dots,x_n))$$

© H. Peter Gumm, Philipps-Universität Marburg

LOOP-berechenbar = Primitiv rekursiv



- Eine Funktion ist genau dann p.r. wenn sie LOOP-berechenbar ist.
- Jede p.r.-Funktion ist total.
- Die Umkehrung gilt nicht. Wir suchen ein Gegenbeispiel
- Wir wissen schon: Die Funktion muss sehr komplex sein – jedenfalls von mehr als polynomialem Aufwand !
- Berühmtes Beispiel: Ackermannfunktion

© H. Peter Gumm, Philipps-Universität Marburg

Ackermannfunktion

Wilhelm Ackermann
1896-1962

- Idee: Betrachte die Serie
  - mult(x,0) = 0
  - mult(x,y+1) = add(x,mult(x,y))
  - exp(x,0) = 1
  - exp(x,y+1) = mult(x,exp(x,y))
  - hyp(x,0) = 2
  - hyp(x,y+1) = exp(x,hyp(x,y))
  - ...

- Wir können die Serie parametrisiert definieren
  - a(k,x,0) = k
  - a(0,x,y) = add(x,y)
  - a(k+1,x,y+1) = a(k,x,a(k+1,x,y))
- Diese Funktion ist berechenbar und total. Ihre Definition folgt nicht dem p.r.-Schema.
- Ackermann zeigte: Nicht p.r.

© H. Peter Gumm, Philipps-Universität Marburg

Ackermann-Péter Funktion

Rózsa Péter
1905-1977

- Von Rózsa Péter vereinfachte Variante der Ackermannfunktion:
  - $ack(0,y) = y+1$
  - $ack(x+1,0) = ack(x,1)$
  - $ack(x+1,y+1) = ack(x,ack(x+1,y))$

- ack ist total und berechenbar, z.B.:

$$
\begin{aligned}
ack(3,3) &= ack(2,ack(3,2)) \\
&= ack(2,ack(2,ack(3,1))) \\
&= ack(2,ack(2,ack(2,ack(3,0)))) \\
&= ack(2,ack(2,ack(2,ack(2,1)))) \\
&= \ldots \\
&= 61
\end{aligned}
$$

- ack wächst extrem schnell, z.B.:

$ack(4,2) \approx 10^{21000}$

$ack(4,4) \approx 10^{10^{10^{21000}}}$

© H. Peter Gumm, Philipps-Universität Marburg

ack dominiert jede p.r. Funktion

- Satz: Für jede p.r.-Funktion $f: N^k \to N$ gibt es ein k, so dass für alle $x_1,..,x_k \in N$

$$f(x_1,..,x_k) < ack(k, x_1+...+x_k)$$

- Folgerung: ack wächst schneller als jede p.r. Funktion

- Korollar: ack ist total, berechenbar, aber nicht primitiv rekursiv.

  - Beweis des Korollars: Wäre ack p.r., dann auch $f(n)=ack(n,n)$.
  - Dann ex. $k \in N$. $f(n) < ack(k,n)$. Es folgt

    - $ack(k,k) = f(k) < ack(k,k)$.

Widerspruch.

© H. Peter Gumm, Philipps-Universität Marburg

Beweisvorbereitung: Einige Lemmata

1. $ack(1,y) = y+2$
2. $ack(2,y) = 2y+3$
   - Beweise: Induktion über y
3. $y < ack(x,y)$
   - Induktion über x mit Nebeninduktion über y
4. $ack(x,y) < ack(x,y+1)$
   - Ind. über x
5. $ack(x,y+1) \leq ack(x+1,y)$
   - Ind. über y

- $ack(x,y) < ack(x+1,y)$
  1. aus dem Vorigen

© H. Peter Gumm, Philipps-Universität Marburg

ack dominiert sich selbst

Lemma: Zu a_1, a_2 gibt es stets ein c mit: $ack(a_1,y) + ack(a_2,y) < ack(c,y)$

Beweis: Setze $d = max(a_1,a_2)$, dann gilt

ack(a1 , y) + ack(a2 ,y)	≤ ack(d,y) + ack(d,y)	Monotonie
n	< 2*ack(d,y) + 3	Algebra
n	= ack(2,ack(d,y))	ack(2,y) = 2y+3
n	< ack(d+2,ack(d+3,y))	Monotonie
n	= ack(d+3,y+1)	Definition
n	≤ ack(d+4,y).	ack(x,y+1)≤ack(x+1,y)

Folgerung: Zu a_1, ..., a_n gibt es c mit: $ack(a_1,y)+\ldots+ack(a_n,y) < ack(c,y)$

© H. Peter Gumm, Philipps-Universität Marburg

Der Ackermann'sche Beweis

Satz: Für jede primitiv rekursive Funktion f gibt es ein a, so daß
$f(y_1,\ldots,y_n) < ack(a, y_1+\ldots+y_n)$.

Beweis: Für die Basisfunktionen ist die Behauptung klar.

Wir betrachten hier nur ein- bzw. zwei-stellige Funktionen.
Seien g und h primitiv rekursiv mit $g(y) < ack(a,y)$ und $h(y) < ack(b,y)$.

Komposition: $g(h(y)) < ack(a,h(y))$
$\leq ack(a,ack(b,y))$
$< ack(m,ack(m+1,y))$, mit $m=max(a,b)$
$= ack(m+1,y)$.

Primitive Rekursion: Sei $f(0,y) = g(y)$ und $f(x+1,y) = h(f(x,y),x,y)$, und
$g(y) < ack(a,y)$, sowie $h(x,y,z) < ack(b,x+y+z)$ nach Induktionsvorraussetzung.

Es folgt $g(y) + y < ack(a',y)$ sowie $h(x,y,z) + x+ y +z < ack(b',x+y+z)$, für
$a' > a$, $b' > b$. Für $e = max(a',b')+1$ zeigen wir nun durch Induktion nach x:

$$f(x,y) + x + y < ack(e,x+y).$$

© H. Peter Gumm, Philipps-Universität Marburg

Minimalisierung

- Primitive Rekursion liefert nicht alle berechenbaren Funktionen
  - Nicht einmal alle totalen
    - Wohl aber alle „effizient berechenbaren“
  - Wir brauchen ein weiteres Rekursionsprinzip
  - Was ist noch nicht erfasst ?
- Lösung suchen durch Ausprobieren
  - Gegeben $f: N^k \rightarrow N$ und $x_2,\ldots,x_k \in N$
  - Finde kleinste Zahl n mit $f(n,x_2,\ldots,x_k)=0$
  - $min(f)(x_2,\ldots,x_k) = min\{n \in N \mid f(n,x_2,\ldots,x_k) = 0\}$
- Berechenbar ?
  - Selbst wenn f total ist, kann min(f) partiell sein
  - Wenn f partiell ist, wie sollen wir min(f) berechnen ?
    - Ausprobieren : n=0, n=1, n=2, ...
    - Was aber, wenn $f(1,x_2,\ldots,x_k) = 0$ aber $f(0,x_2,\ldots,x_k) = \perp$?

© H. Peter Gumm, Philipps-Universität Marburg

Der μ-Operator

- Sei $f: N^{k+1} \rightarrow N$ eine berechenbare Funktion. Definiere

$$\mu(f)(x_1,\ldots,x_k) := \begin{cases} \min\{ n \mid f(n,x_1,\ldots,x_k) = 0 \} & \text{falls } \forall\, k \leq n.\ f(k,x_1,\ldots,x_k) \neq \bot \\ \bot & \text{sonst.} \end{cases}$$

- $\mu(f) : N^k \rightarrow N$ ist berechenbar
- $\mu(f)$ kann partiell sein, selbst wenn f total war.
- Beispiel:
  Seien t(x,n) = not(x*(n/x) – n) und gv(n,x,y) = t(x,n) and t(y,n).
  Dann gilt:

  $$\mu(gv)(x,y) = ggT(x,y)$$

  Sei gt(n,x,y) = t(min(x,y)-n,x) and t(min(x,y)-n,y)
  Dann gilt

  $$\min(x,y) - \mu(gt)(n,x,y) = ggT(x,y)$$

© H. Peter Gumm, Philipps-Universität Marburg

μ-rekursive Funktionen

- Basisfunktionen
  - $f(x) = k$ — Konstante $k \in N$
  - $\pi^n_i(x_1,\ldots,x_n) = x_i$ — i-te Projektion
  - $succ(x) = x+1$ — Nachfolger
- Abschluss unter
  - Komposition (Einsetzen)
  - primitives Rekursion
  - μ-Operator

© H. Peter Gumm, Philipps-Universität Marburg

μ-Rekursion ist Berechenbarkeitskonzept

- Die μ-rekursiven Funktionen sind genau die partiellen berechenbaren Funktionen

Beweisidee (Äquivalenz zu While-Programmen):

- μ-Operator durch ein While-Programm zu implementieren ist einfach. Folglich ist jede μ-rekursive Funktion While-berechenbar

- Umgekehrt sei f durch ein While-Programm gegeben. Nach Kleene sogar als

S_1 ; While B(x) S_2

wobei kein While in S_1 bzw. in S_2 vorkommt.

Mit μ-Operator finde Anzahl n der Schleifendurchläufe.
Ersetze while B do P durch For k:=0TO n DO P (Siehe nächste Folie)

Jetzt haben wir ein LOOP-Programm, in dem ein μ-Operator vorkommt. Wie vorher können wir das durch eine primitiv rekursive Funktion simulieren, in der jetzt ein μ-Operator vorkommt, die also insgesamt μ-rekursiv ist.

© H. Peter Gumm, Philipps-Universität Marburg

WHILE ist μ-rekursiv

Erinnerung

$g(0,x_1,\ldots,x_n) = (x_1,\ldots,x_n)$
$g(k+1,x_1,\ldots,x_n) = f_P(g(k,x_1,\ldots,x_n))$

- while B do P
  - Sei $g(n,x_1,\ldots,x_n)$ die semantische Funktion von FOR k:=0 TO n DO P
  - Sei $f_B(x_1,\ldots,x_n)$ die semantische Funktion von B, dann gilt

$$h(n,x_1,\ldots,x_n) = f_B(g(n,x_1,\ldots,x_n)) = \begin{cases} 1, & \text{falls B wahr nach n mal P} \\ 0, & \text{sonst} \end{cases}$$

  - $\mu(h)(x_1,\ldots,x_n)$ = benötigte Schleifendurchläufe in while B do P
  - Folglich ist while B do P = FOR k:=0 TO $\mu(h)(x_1,\ldots,x_n)$ DO P
  - Folglich ist while B do P durch eine μ-rekursive Funktion berechenbar

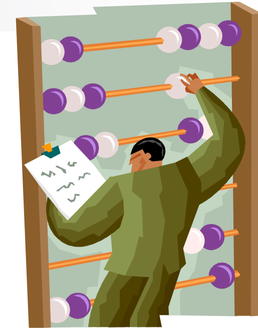
© H. Peter Gumm, Philipps-Universität Marburg

Inhalt

1. Turingmaschinen
   - Definition und Beispiele
   - Turing-Post Programme
   - Registermaschinen
   - Äquivalenz
2. Programmierparadigmen
   - GOTO
   - LOOP- und WHILE-Sprachen
   - Äquivalenz von GOTO und WHILE
   - Churchsche These
3. Rekursive Funktionen
   - Primitive Rekursion
   - Äquivalenz zu LOOP
   - μ-Rekursion
   - Äquivalenz zu While
4. Halteproblem
   - Aufzählbarkeit u. Entscheidbarkeit
   - Halteproblem
   - Nichtberechenbare Funktionen
   - Satz von Rice

© H. Peter Gumm, Philipps-Universität Marburg

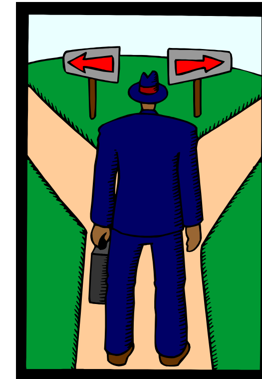
Aufzählbarkeit



- Eine Menge U heißt aufzählbar, falls es eine surjektive totale berechenbare Funktion $f:N \rightarrow U$ gibt.
- U ist genau dann aufzählbar, wenn es einen Algorithmus gibt, der genau alle Elemente von U produziert:
  - Wir nehmen dazu eine Schreibanweisung „Write <n>" an:
    n := 0 ; While true Do { Write(f(n)); n := n+1 }
- Beispiele:
  - Die Menge aller Primzahlen ist aufzählbar.
  - Die Menge aller Teilworte von π ist aufzählbar
  - Die Menge aller n mit Ulam(n)=1 ist aufzählbar

© H. Peter Gumm, Philipps-Universität Marburg

Entscheidbarkeit



- Eine Teilmenge U $\subseteq$ N (analog U $\subseteq \Sigma^*$) heißt entscheidbar, wenn ihre charakteristische Funktion

$$\chi_U(n) := \begin{cases} 1 & \text{falls } n \in U \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist

- U heißt semientscheidbar, wenn

$$\partial_U(n) := \begin{cases} 1 & \text{falls } n \in U \\ \bot & \text{sonst} \end{cases}$$

berechenbar ist

- Satz: U ist genau dann entscheidbar, wenn U und N-U semientscheidbar.
  - Beweis : klar

Aufzählbar = Semi-entscheidbar

- Satz: Eine Menge ist genau dann aufzählbar, wenn sie semi-entscheidbar ist.

- Beweis: U sei durch Algorithmus A aufzählbar. Gegeben k, warte, bis A k ausgibt, dann halte mit Ergebnis 1. Das ist ein Semi-entscheidungsverfahren.

- Jetzt sei U semientscheidbar. $D_U(i)$ sei das zugehörige While-Programm, d.h. $D_U(i)$ hält $\Leftrightarrow i \in U$, Betrachte $D_U(n,k)$:

$D_U(i)$
```
S1;
While B do S2
```

$D_U(n,i)$
```
S1 ;
For k := 0 to n
    If B do S2 ;
If not(B) then Write(i)
```

- Der folgende Algorithmus zählt jetzt U auf:
  - n := 0; WHILE true DO { FOR k := 0 TO n DO $D_U(n,k)$; n := n+1 }

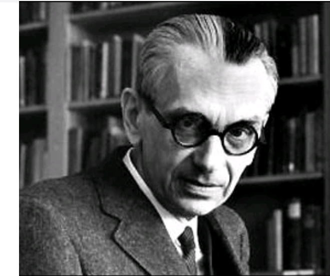
Aufzählbarkeit

- Sei U⊆N (bzw. U ⊆Σ*)
  - Äquivalent
    - U ist semi-entscheidbar
    - U ist Bild einer totalen berechenbaren Funktion
    - U ist Definitionsbereich einer partiellen berechenbaren Funktion
  - Es sind äquivalent
    - U ist entscheidbar
    - U und N-U sind aufzählbar
    - U ist monoton aufzählbar

U heißt monoton aufzählbar, falls die Elemente von U in der natürlichen Reihenfolge aufgezählt werden.

© H. Peter Gumm, Philipps-Universität Marburg

Gödelisierung



Kurt Gödel

1906-1978

- Wähle irgendeins der Algorithmenkonzepte
  - z.B. Registermaschinen
- Wir können systematisch alle $w \in ASCII^*$ auflisten und jeweils testen, ob es sich um ein RM-Programm handelt
- Ergebnis: Auflistung $P_1, P_2, P_3, \ldots$ aller RM-Programme
- Jedes RM-Programm P ist eines der P_i.

  i ist dann die Gödelnummer von P ($i = GN(P)$)
- Gegeben ein P_i und ein beliebiges $k \in N$, dann berechnet P_i eine k-stellige partielle Funktion

$$\varphi_i^{(k)} : N^k \rightarrow N :$$

Wie? Fülle $x_1, .. x_k$ in Register c[1],...,c[k], dann starte P_i .

- Falls P_i hält, ist $\varphi_i^{(k)}(x_1, .. x_k) = c[1]$
- Falls P_i nicht hält, ist $\varphi_i^{(k)}(x_1, .. x_k) = \perp$

- Jede partielle berechenbare k-stellige Funktion f erscheint (mehrfach) als ein $\varphi_i^{(k)}$.

  i ist dann eine Gödelnummer von f ($i \in GN(f)$)

© H. Peter Gumm, Philipps-Universität Marburg

Aufzählbarkeit der part. berechenbaren Fkt

- Die partiellen berechenbaren (k-stelligen) Funktionen lassen sich aufzählen
  - Systematische Auflistung aller RM-Programme
    - $P_1, P_2, P_3, \ldots$
  - Liefert Aufzählung aller (k-stelligen) partiellen berechenbaren Funktionen
    - $\varphi_1^{(k)}, \varphi_2^{(k)}, \varphi_3^{(k)}, \ldots$
  - Jede Funktion kommt in dieser Aufzählung mehrfach (sogar unendlich oft) vor
    - Zu jeder partiellen berechenbaren Funktion f gibt es (unendlich) viele P_i, die f berechnen

© H. Peter Gumm, Philipps-Universität Marburg

Totale Funktionen nicht aufzählbar

- Die totalen berechenbaren (k-stelligen) Funktionen sind nicht aufzählbar
  - Widerspruchsbeweis (Cantor's Trick):
  - Angenommen $\psi_1, \psi_2, \psi_3, \ldots$ sei eine Aufzählung aller totalen berechenbaren 1-stelligen Funktionen
  - Betrachte die Funktion

$$\Delta(n) := \psi_n(n) + 1$$

  - Δ ist offensichtlich berechenbar.
  - Aber Δ ist verschieden von allen ψ_i , kommt also nicht in der Liste vor. Widerspruch !!

© H. Peter Gumm, Philipps-Universität Marburg

Halteproblem

- Allgemeines Halteproblem
  - Gegeben
    - Ein Programm P
    - Ein Input n
  - Gefragt:
    - Hält das Programm P mit Input n
  - Konkret:
    - Sei $A(m,n) := \begin{cases} 0 & \text{falls } \varphi_m(n) = \bot \\ 1 & \text{sonst} \end{cases}$
    - Ist A(m,n) berechenbar ?

- Spezielles Halteproblem
  - Gegeben
    - Programm P_m (Programm mit Gödelnummer m)
  - Gefragt
    - Hält P_m mit Input m , das heißt:
    - „Ist D(m) := A(m,m) berechenbar ?“

© H. Peter Gumm, Philipps-Universität Marburg

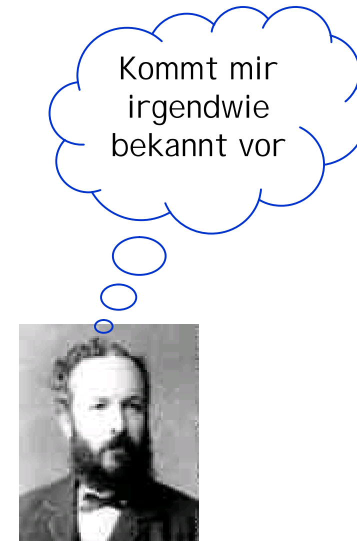
Unlösbarkeit des Halteproblems

- Das spezielle Halteproblem ist nicht lösbar

- Folgerungen
  - Das allgemeine Halteproblem ist nicht lösbar.
  - A(m,m) ist totale aber nicht berechenbare Funktion.
  - $H = \{n \in N \mid A(n,n) = 0\}$ nicht semi-entscheidbar.

- Beweis: Angenommen A(n,n) wäre berechenbar.
- Bastele neuen Algorithmus Δ:

$\Delta(n)$
```
if A(n,n) = 1
    then { while true do { } }
    else return 1
```



- Δ habe Gödelnummer k, d.h. $\Delta = P_k$. Was liefert $\Delta(k)$?

- $\Delta(k)=1 \Leftrightarrow A(k,k)=0 \Leftrightarrow \Delta(k)=\perp$ Widerspruch!

© H. Peter Gumm, Philipps-Universität Marburg

Satz von Rice

- Sind die folgenden Fragen für Algorithmen entscheidbar ?
  - … ob sie einen Output generieren ?
  - … ob sie bei einem bestimmten Input halten ?
  - … ob zwei die gleiche Funktion berechnen ?
  - … ob sie bei mindestens einem Input halten ?
  - …

- Henry Gordon Rice: Nein, nein, und nochmals nein !!
  - All das folgt sofort aus dem folgenden Satz von Rice:

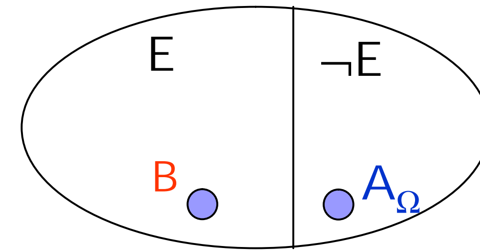
> Satz: „Jede nichttriviale semantische Eigenschaft von Algorithmen ist unentscheidbar

- Eigenschaft E von Algorithmen heißt semantische Eigenschaft, falls :

$$\frac{P_m \text{ erfüllt } E\,,\quad \varphi_m = \varphi_n}{P_n \text{ erfüllt } E}$$

© H. Peter Gumm, Philipps-Universität Marburg

Satz von Rice



- Beweis des Satzes von Rice:

  E sei nichttriviale semantische Eigenschaft für Algorithmen.

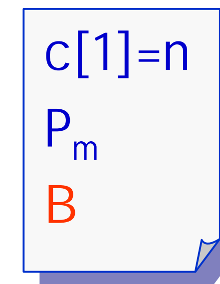
  Sei A_Ω = while true do { }.

  Wir können annehmen: A_Ω erfüllt nicht E
  (andernfalls betrachten wir ¬E).

  Sei B ein Algorithmus mit B erfüllt E (existiert, da E nichttrivial).

  Angenommen, E wäre entscheidbar, dann könnten wir das Halteproblem A(m,n) folgendermaßen lösen:

  - Gegeben m,n, erzeuge den Algorithmus:
  - Er ist entweder äquivalent zu A_Ω oder zu B !
  - Entscheide, ob er E erfüllt.
    - Wenn ja: $P_m(n)$ hält
    - Wenn nein: $P_m(n)$ hält nicht.



Übersicht

berechenbar

nicht berechenbar

Funktion mit polynomialer Komplexität

totale Funktionen

primitiv rekursiv
= LOOP-berechenbar

partielle Funktionen

Ackermann-funktion

semantische Eigenschaft –
z.B. Halteproblem

© H. Peter Gumm, Philipps-Universität Marburg